Object-Oriented Programming in Java

Notes for the Object-Oriented Programming Courses at Politecnico di Torino

Marco Torchiano Stefano Mancini

2025-12-22

Table of contents

Preface

This books contains a collection of the materials presented during the lectures of the Object-Oriented Programming courses at Politecnico di Torino.

This is not inteded as a complete description of the Java programming language, for that purpose there are severe complete books as well as the official online resources.

The goal of this book is to provide a progressive introduction to the Java language and its main important characteristics.

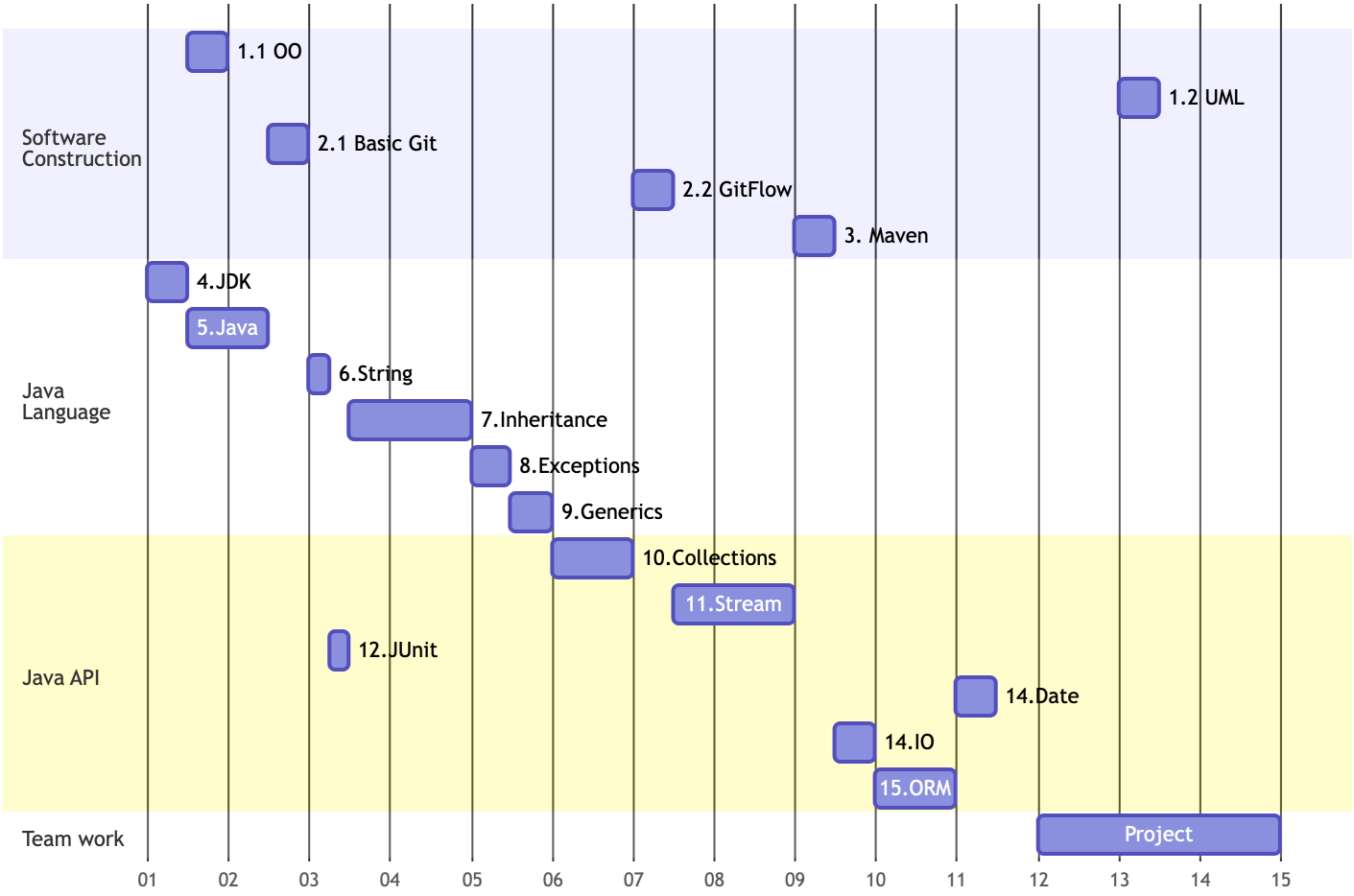
The course covers three main areas that are reflected into the parts of this book:

- Software Construction fundamentals: provides a few – language-agnostic – software engineering concepts that represent the basis for a disciplined software construction process. In particular it focuses on the Object-Oriented approach, configuration management, and build management.
- Java Language definition: describes the basic syntax of the Java language – from foundations such as classes and objects, to inheritance and generic types –
- Java APIs: introduces the main standard libraries and APIs that are part of the Java ecosystem at large, starting from the collections framework and including the JUnit test framework.

The order of the chapters follows this macro-classification, although for practical reasons the lectures of the course treat the topics in a different order. Considering a 14 weeks semester, a rough subdivision of the topics is:

Week	Topic
W1	Object-Oriented Paradigm and Java Environment
W2	Java Basic Features and Basic Git
W3	Java Characters and Strings and JUnit
W4	Inheritance
W5	Generics and Exceptions
W6	Collections framework
W7	GitFlow
W8	Stream API
W9	Java IO and Build management
W10	Object-Relational Mapping
W11	Date and Time
W12	Team project presentation
W13	UML Design
W14	Project review

A more refined outline is reported in the following GANTT diagram:



License

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Colophon

This is a book created from markdown and executable code using Quarto books.

See Knuth (1984) for additional discussion of *literate programming* that is the general approach of mixing code and text descriptions.

Part I.

Software Construction

1. Object Oriented Paradigm (Draft)

We ough to understand how we move from the procedural programming approach to an object-oriented approach, that is, understanding the motivations and the fundamental characteristics of the object-oriented paradigm.

1.1. Object Oriented Paradigm

There are several programming paradigms, i.e. the model of thinking about programs. A paradigm defines how computation is structured, how state and control flow are handled, and what abstractions can be used. In prinicple you can adopt a paradigm event if the language is not designed for it, but such task requires a much larger effort than when using a language that supports the required paradigm.

The main programming paradigm are:

- Procedural (Pascal, C,…)
- Object-Oriented (C++, Java, C#,…)
- Functional (LISP, Haskell, SQL,…)
- Logic (Prolog)

Many languages are multi-paradigm, allowing a mix of styles. Historically, new programming paradigms have emerged together with languages offering support for such paradigms, thus providing proof of concept for the paradigm.

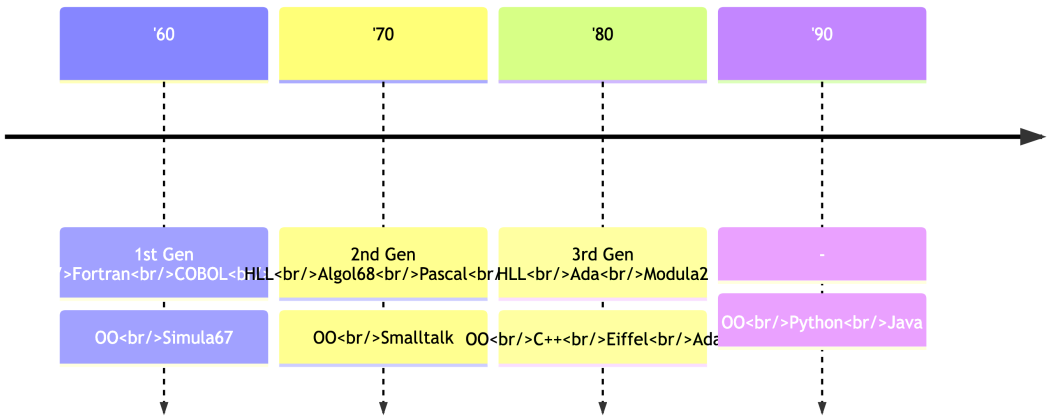


Figure 1.1.: Languages timeline

The first high-level programming languages – mainly Fortran and COBOL in the 1960s – were procedural. Precisely because they were high-level languages, they allowed you to define variables and structured data types.

Later, toward the late 1960s and early 1970s, a second generation of high-level procedural languages emerged: C is the main example, but there were also Pascal and ALGOL 68. These introduced **structured programming**(Dijkstra 1968) -— block-structured control constructs -— and more structured data typing, with conversions and compatibility checks among variable types when used together.

Subsequently, in the late 1970s and early 1980s, further high-level procedural languages appeared—Ada among the best-known—which defined the notions of modules, concurrent programming, and exceptions. In parallel with this main procedural line, there was also a series of object-oriented languages. The first was Simula 67 (as the name suggests, developed in 1967), the first programming language that truly had the main constructs we now associate with object-oriented languages. Later came Smalltalk in the early 1970s, which saw significant use. Smalltalk had some limitations, especially in terms of efficiency, but it helped spread the idea and approach of object-oriented programming.

Finally, in the 1980s, C++ emerged, along with a couple of other languages (such as Eiffel) and Ada's evolution (from an initially procedural language). In the 1990s, a second generation of object-oriented languages appeared; Java is the main one, but we also have others like Python, and then toward the late 1990s C#, a fairly widespread object-oriented language.

The object-oriented approach introduces the concept of **object**. It is a new language-level construct (i.e. something that is part of the language syntax) that groups together the *data* and the *functions* that operate on them. An object is an entity with an *interface* -— the set of operations that it accepts and that correspond to some function in the object – and a *state* – the values stored in the data is contains – that can be changed by the functions.

A fundamental aspect of objects is that they hide data. **Information hiding** is the principle of hiding data inside an object so that it can be seen and modified only by the functions inside that object.

1.1.1. Object and Classes

The fundamental idea of the object-oriented approach is that we define *objects* representing well-known domain concepts with a clear role in the problem domain. Beyond objects, the object-oriented approach includes *classes*.

It is important to understand that when dealing with OO development, there are two distinct abstraction levels that we can call abstract level and concrete level. At the abstract level we talk about concepts, entities, categories, types; at the concrete level we talk about instances, elements, objects, examples, occurrences. OO moves between these two with the terms class (abstract, design-time) and object (concrete, runtime). At design time we define classes; at runtime we observe objects interacting, containing data, and operating on their own data.

Level	Examples
Abstract	Concept Entity Class Category Type

Concrete	Instance Item Object Example Occurrence

It is a common case to have many similar objects embodying the same idea (e.g., product). A **class** defines the common characteristics of all objects of that kind, it constitutes a higher-level concept with respect to object. The passage linking class and object is called *instantiation*: a class can be instantiated to create objects, and all objects instantiated from the same class share the same characteristics. E.g., for a `Product` class, common features might be having a `name` and a `price`. Objects are created – at run-time – as instances of classes – defined at compile-time –. The class defines the structure of an object an therefore it represents the type of the object. A class defines how an object is built through a constructor function and initialization is performed automatically by the language.

A class declaration is also a type definition, typically no memory is allocated until an object is created from the class.

The creation of an object is called instantiation. The created object is often called an instance of the class. There is no limit to the number of objects that can be created from a class. Each object is independent and has its own identity, thus interacting with one object doesn't affect the others.

Once a class is identified to represent a specific concept, its structure – how its instances are laid out – must be detailed. The core elements in a class are the **attributes** (just like variables in any language: they have a name and a type). The class defines the names and types; each object holds values for each attribute. The set of values that are associated to the attributes if an object constitute the state of an object.

Class vs. Object:

- Class (the description of object structure, i.e. type):
  - Data structure (attributes or fields or properties)
  - Behavior (methods or operations)
  - Creation methods (constructors)
- Object (class instance)
  - State and identity

An engineering approach would require that, given a system, with components and relationships among them, we shall: first identify the components, then define the component interfaces, define how components interact with each other through their interfaces. And in general minimize relationships among components.

The **interface** of a component is composed of the set of messages that an object can receive. Each message is mapped to an internal procedure within the object. The object is responsible for the association (message -> function). Any other message is illegaland results in an error.

The interface encapsulates the internals and exposes a standard boundary through with interactio with other objects occurr. The interface of an object is simply the subset of methods that other "program parts" are allowed to interact with.

Encapsulation lead to several benefits:

- simplified access: to use an object, the user need only comprehend just the interface. No knowledge of the internals are necessary;
- self-containmen: once the interface is defined, the programmer can implement the interface (i.e. define the the object internals) without interference from others;
- ease of evolution: implementation can change at a later time without affecting any other part of the program (as long as the interface doesn't change)
- single point of change: any change in the data structure means modifying the code in one location, rather than code scattered around the program (which is error prone)

Classes can also be related to each other: for instance a receipt contains items/lines; each item refers to a product; products are contained in a price list.

1.1.2. Inheritance

In OOP, classes can also support *inheritance*: from a base class you can define a more specific class that inherits all the features of the base class with a sort of virtual copy-and-paste.

Inheritance enables two important related features: - **polimorphism** is the ability to manage uniformly different object types that are able to respond to the same operation in their own specific way. - **dynamic binding** is the process of determining at runtime which method implementation to invoke based on the actual type of the object, i.e. determining the specific way an object respond to an operation.

1.1.3. OO Summary

In summary classes

- represent high level concepts, often taken from problem domain;
- are instantiated into Objects;
- define common features of Objects;
- can be related to each other, thus defining links and communication patterns among their instances (i.e. objects);
- can be defined through inheritance, where specific classes inherit from general ones.

The adoption of the object-oriented approach was motivated by programs getting too large to be fully comprehensible by any person. A requirement emerged for a way of managing very-large projects. Thus the Object Oriented (OO) paradigm allows programmers to (re)use large blocks of code without knowing all the picture. OO makes code reuse a real possibility and simplifies maintenance and evolution. Remember that most software cost is not in initial development but in maintenance, therefore any approach that reduces maintenance is a major advantage.

The actual benefits of adopting OO only occur in larger programs. Analogous to structured programming: for programs with e.g. 30 lines of codde, *spaghetti* is as understandable and faster to write than a well structured OO code. While for programs with thousands of lines of code, *spaghetti* is incomprehensible, probably doesn't work, and it is not maintainable. Only programs with more than a few hundreds lines of code really benefit from OO.

Historically, object-oriented language construction evolved untile they achieve what we call today the object oriented paradigm.

- The earliest were *object-based* (e.g. Ada): you could build an object as a set of related data and functions, but without the notion of class as a single definition for all objects sharing the same structure.
- The next step *class based* (e.g., CLU) introduced classes, separating the high-level design concept from the low-level runtime entities.
- The following step gave us fully *object-oriented* languages (e.g., Simula, Python) where objects come from classes and classes can inherit from each other, adding useful properties.
- A subset are *strongly typed OO* languages (C++, Java), where data types are precisely defined and the compiler can perform many consistency checks.

  Think of C and Java: every variable has a well-defined type. (C is strongly typed though not object-oriented; Java is both.) Other languages, like Python, are dynamically typed: you define variables without fixed types; a variable can hold values of different types at different times; this prevents the compiler from doing certain checks; inconsistencies surface at runtime instead of compile time. In general, strong typing lets us catch potential errors earlier.

1.2. Example

To better understand what it means to develop in a procedural style versus an object-oriented style we can consider an simple example concerning a cash register.

1.2.1. Case study: cash register

The requirements requirements for the (simplified) cash register software are:

A cash register emits purchase receipts. A receipt is made up of a list of items. Each item corresponds to a product that has a name and a price. The information about the products is stored in a price list.

Any time a new product code is entered the corresponding item is added to the receipt. After the last item is entered, a list of all items (with product name and price) is printed, together with the total sum.

Example of interaction with the cash register

Input:

```
13 # <1>
57 # <1>
123 # <1>
0 # <2>
2 # <3>
10 # <4>
```

① product codes
② the `0` at the end represents the end of items and signals the list of items is complete
③ the type of payment (2 = cash)

④ the amount of cash received (10€)

Output:

```
Receipt:
    Banana: €   0.99
     Pasta: €   4.10
     Bread: €   3.49
--------------------
No items: 3

     Total: €   8.58

Cash provided: €  10.00
         Rest: €   3.53
```

1.2.2. Procedural solution

Let us consider a procudural implementation in C. First of all we need to think about what data structures and functions are needed to implement the program that.

Concerning the data structures, we need:

- for the price list: an array of numbers -— the prices —- paired with an array of strings – the product names -—; let's assume, for simplicity, a product's code is the same as its index in these arrays.
- for the items: an array of integers containing the product codes that represent the items included in the receipt, and a count of how many items are there on the receipt.
- for the payment: a structu

As far as the operations (functions) are concerned, what is needed is essentially:

- a function to read the codes from the cashier,
- a function to add product codes to the receipt;
- a function to process the payment;
- a function to print the receipt at the end;
- a function to initialize everything.

We also need a main function that initializes, then loops reading codes until it gets a special code that marks the end of the receipt, adding all codes; once the last product is reached, it prints the receipt — i.e., what we see on the printout.

So here is the draft of a possible solution, without the code details:

```
float prices[MAX_LIST];
char* names[MAX_LIST];
int receipt[MAX_RCPT];
int n_items;
struct {
```

```
   int type;
   double amount;
   union {
      struct { int card_type; } credit;
      struct { double cash, rest;  } cash;
    };
} payment;
void add(int); /* add an item to receipt */
void print(); /* print receipt */
void init(); /* initialize */
int read(); /* read the next item code */
void process_payment(); /* process payment */
int main(){
   init();
   int code;
   while( (code = read()) ){ add(code); }
   print();
}
```

There are several limitations in this solution:

- there is no any intrinsic link between the two arrays -— `prices` and `names` –. As programmers we consider them to be "parallel" so we give them the same size, but there's no guarantee enforced by the language;
- there is no link between the item count (`n_items`) and the `receipt` array, in practive there is no control over size, apart from the code that we explicitly write;
- it is not clear which function is allowed to operate on the data: essentially the functions `add`, `print`, and `init`. The receipt, item count, and the functions are strongly related to each other, but in the code we don't see this relationship. There's nothing in the language syntax to make them exist together as a single cohesive unit.
- looking at `main`, we can observe that it calls `init` at the start. If one developer wrote the functions and data, and someone else used them, how can we be sure initialization happens correctly *before* the rest of the code uses the data, it could be in the documentation but we know real programmers don't write documentation.

The above solution can be modeled with a diagram that represents functions (in purple) with arrows indicating calls among them, and data (in white) that the various functions read or write. There's clearly no construct that binds these three functions and two data structures into a single cohesive unit.

All these details limit the approach and are sources of potential errors when we write a program procedurally.

1.2.3. Object-Oriented Solution

We can apply the principle of object-orientation to migrate the procedural solution into an OO solution.

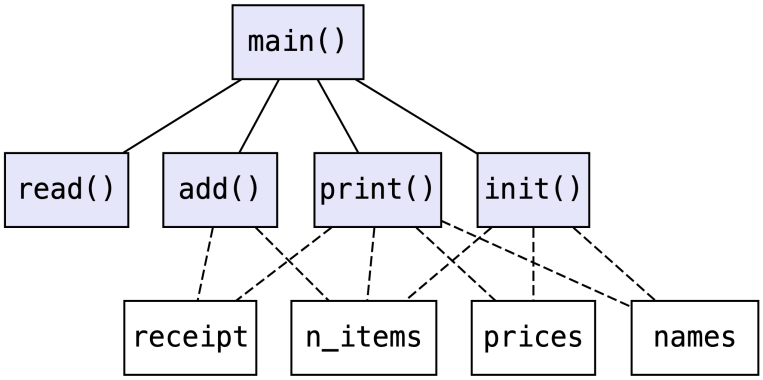


Figure 1.2.: Elements of a procedural solution and their relations

First of all we can apply the principle of **encapsulation**.

Bringing together together related code and data, i.e. `init()` + `add()` + `print()` + `receipt` + `n_items`and ecapsulating them into an object called *Receipt* can be modeled as represente in Figure 1.3.

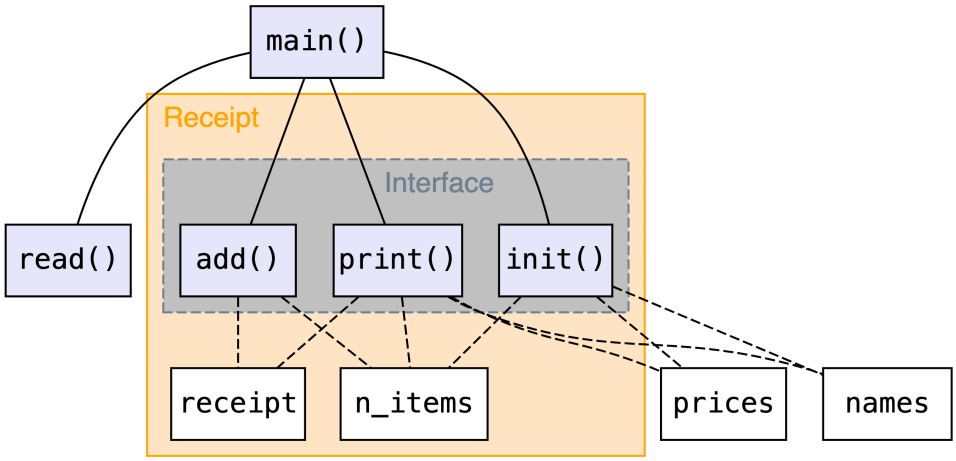


Figure 1.3.: Elements of a procedural solution and their relations

You can't arbitrarily add a new function elsewhere that accesses those data, bypassing the three main functions (add/print/init) that are meant to manage them.

Obviously, we can identify other correlated chunks of data. Besides the receipt + item count + three methods, we could have another object that groups the price list/catalog.

It is possible go further, identifying semantically consistent elements that correspond to the concepts in our problem domain. When discussing receipts with a shopkeeper, we'll talk about *products*, *price list*, *receipts*, and *lines/items* on the receipt. So we might model *receipt items*, *products* (not just a single array, but a pair of data: price and name). Having an object that represents that pair and keeps it together overcomes one of the limits of the procedural approach. The result could be modeled as shown in Figure 1.4.

The object-oriented structure of our cash register software can be represented as shown in Figure 1.5, using a notation called UML Class Diagra:

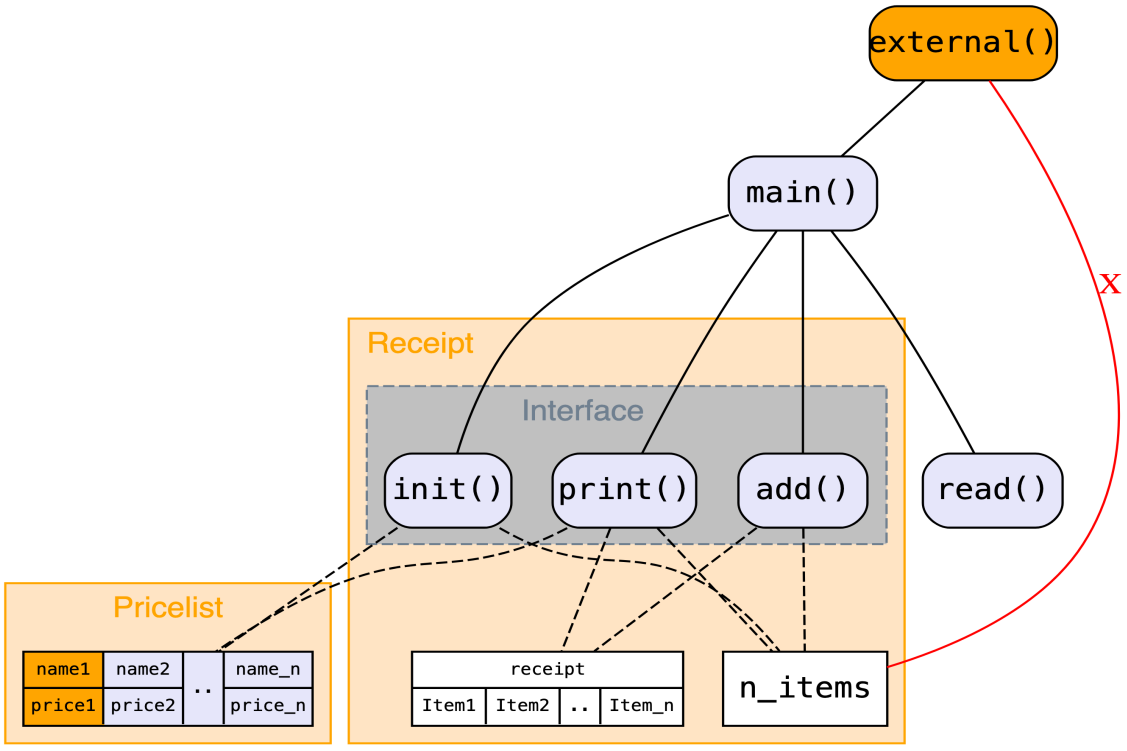


Figure 1.4.: Elements of a procedural solution and their relations

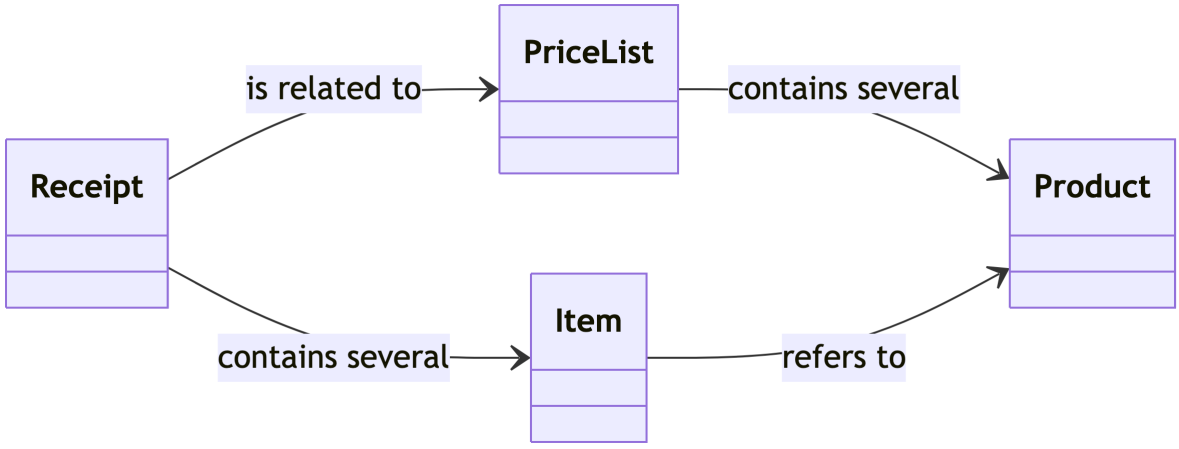


Figure 1.5.: Class diagram of the cash register

In our example we might identify four classes: *Receipt*, *PriceList*, *Item/Line*, and *Product*.`Receipt` is linked to `PriceList`; changing the price list changes values in the receipt. The receipt contains multiple items; each item refers to a product; products are listed in the price list. We've thus moved from a low-level, detail-heavy structure (with things like `MAX_LIST` and `MAX_RECEIPT`, meaningful yet not immediately readable) to a higher-level view with classes. From these classes our program will have objects: one (or more) `Receipt`, one `PriceList`, many `Items`, many `Products`.

For instance, the above example can be extended with the payment for a specific receipt where the payment can be either cash or by credit card. A possible diagram for such exension can be modeled as showin in Figure 1.6.

The class `Payment` represents a general abstraction of payment, while the two derived classes `Cash` and `Card` represent a concrete specialization of that concept. The relation "paid" between `Receipt` and `Payment` means that – by virtue of polymorphism – any receipt can be linked to either a cash or card payment indifferently.

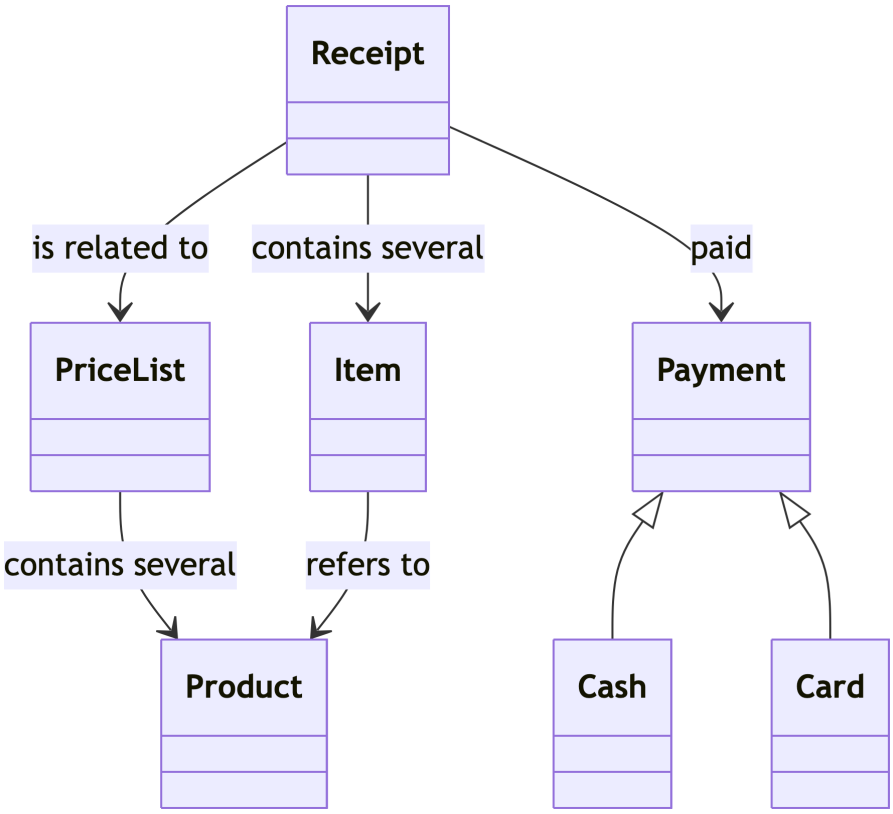


Figure 1.6.: Class diagram for payment

A more detailed UML model of such a solution can be described as follows.

1.3. UML and OO design

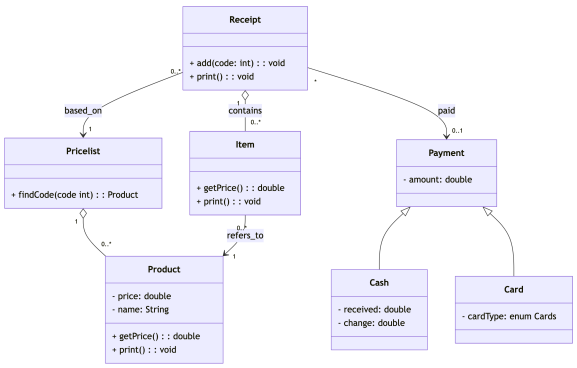


Figure 1.7.: Example of multiplicity in UML

To design a program's structure, given a problem, we first identify the classes (which will become Java code — or code in any OO language) that solve it. To this end, the most commong tool is a graphical notation called **UML** (Unified Modeling Language).

It is a standardized modeling and specification language defined and managed by the Object Management Group (OMG) used to design software with the OO approach. UML provides a graphical notation to specify, visualize, construct, and document an object-oriented system. UML was born as an integration of the concepts of three main methods: Booch (Booch 1993), OMT (Rumbaugh et al. 1994) and RTOOM(Selic, Gullekson, and Ward 1994); it merged them into a single, common and widely used modeling language.

UML defines several diagrams that cover different aspects of software modeling an documentation:

- Class diagrams
- Activity diagrams
- Use Case diagrams
- Sequence diagrams
- Statecharts

1.3.1. UML Class Diagram

The key diagram in OO development is the Class Diagram, it captures:

- Main (abstract) concepts
- Characteristics of the concepts
- Data associated to the concepts
- Relationships between concepts
- Behavior of classes

1.3.2. Classes and objects

A **Class**: represents a set of objects, e.g. facts, things, people, they define

- the common properties – also known as attributes or fields –, i.e. the data they contain,
- the common operations – also called methods – 0they can perform.

An instance of a class is an object, the class represents the type of the object.

Example: in an application for a commercial organization `City`, `Department`, `Employee`, and `Sale` are typical classes.

Classes in UML are represented in UML as rectangles. They are internally divided into three compartments:

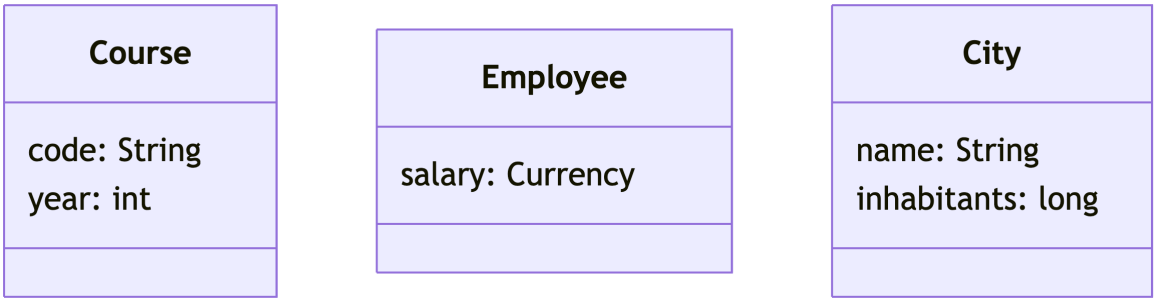
- the top one contains the name of the class
- the middle one contains the properties / attributes
- the bottom one contains the operations / methods

An **Object** is a model of a physical or logical item ex.: a student, an exam, a window. It is characterized by:

- an identity that distinguish it from any other object,
- the state that is the data it contains, (also known as attribute values),
- a set of operations that can invoked on them.

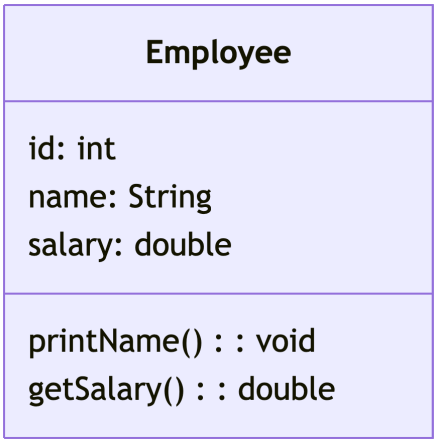
In UML, objects are also rectangles, but unlike classes they typically have a single compartment and are labeled with an object name and optionally the class (e.g., `john : Employee`, `dawin : Department`).

In UML, attributes are listed in the middle compartment (e.g., class Course with attributes `code : String`, `year : int`; class City with `name : String`, `population : int`; class Employee with `salary : Currency`). Note the notation `name : Type` (Pascal-style), which UML adopts.



We also need to define the operations on the data: these are the **methods**. Methods describe how we can operate on internal data; they are analogous to functions in procedural languages (with a name, parameters, and an optional return type).

In UML, methods are listed in the bottom compartment.



We can consider objects communicating by message passing. Not by direct access to object's local data. A message is a service request that is performed by a method.

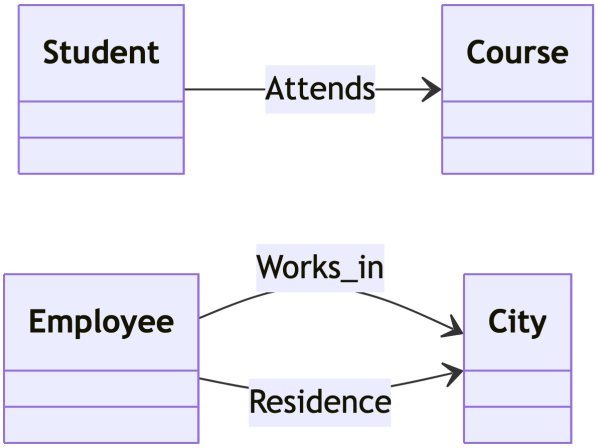
This is an abstract view that is independent from specific programming languages. It allows understanding how a message (the invocation of a method) is decoupled from the actual exceution of the operation.

Often it's helpful to describe these message exchanges with interaction diagrams (sequence diagrams), where we show objects and the messages among them over time.

1.3.3. Associations

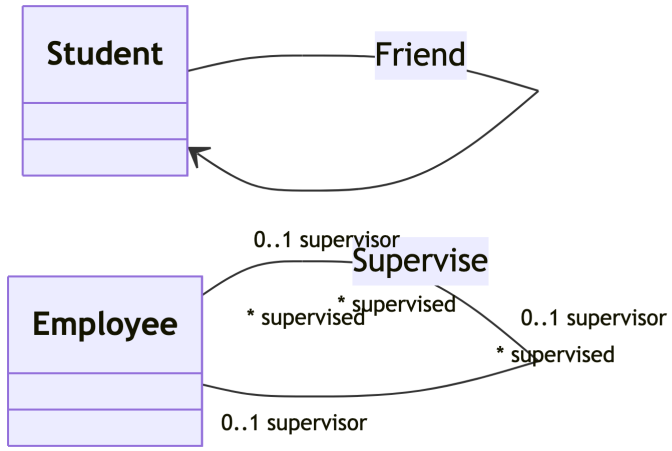
So far we've considered classes in isolation, but real programs have multiple classes related to each other and working together. In UML, links among classes are represented by **associations**, indicating a relationship between two classes. Usually, pairs of objects from those classes will be connected by that association. We define associations at the class level (set-level), not by enumerating every object-level link.

Associations are drawn as lines (possibly with a name and reading direction). For example, Student "attend" Course; Employee "works in" City; some associations are more generic ("residence") and don't need a reading direction.



Associations can be recursive (a class linked to itself), e.g., Student "friend" Student. Such an association doesn't mean a student is a friend of themself; it means some pair of students can be linked by friendship. Some associations are symmetric (friendship), others are not (e.g., supervises between Manager and Employee).

UML lets us name the roles at each end (e.g., `manager` and `employee`).



A link is an occurrence (instance) of an association is a pair made up of the occurrences (instances) of the entities, one for each involved class

Another important concept is multiplicity: how many objects of one class can be linked to how many of the other? Multiplicities are shown as lower..upper bounds at each end (e.g., `0..4`). Often we use `0` or `1` for the lower bound and `1` or `*` ("many") for the upper bound. This matters for implementation choices (a single reference vs. a collection).

Multiplicity should be specified for each class participating in an association

- Minimum: 0 or 1
  - 0 means the participation is optional,
  - 1 means the participation is mandatory;
- Maximum: 1 or *
  - 1: object is involved in at most one link
  - *: each object is involved in many links

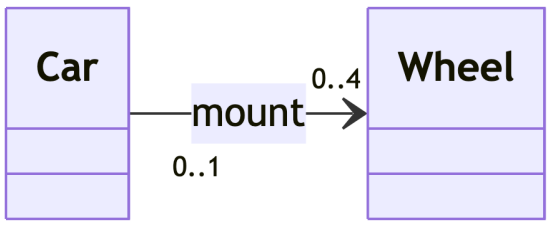


Figure 1.9.: Example of multiplicity in UML

- `0..4` near class *Wheel* means that a car can mount none, up to four wheels.

- `0..1` near class *Car* means that a whell can be mounted on at most one car, possibly none.

An example where the multiplicties are used up to their max is where you a car that is connected to four wheels, as for the object `car` and the wheels `wheel1` .. `whell4` in Figure 1.10. Another option is to have one wheel disconnected therefore (`wheel5`) whose multiplicity w.r.t. the link to car is `0`, and the car has only three wheels mounted (multiplicity `3`). Another confituration of objects that is compatible with the multiplicity defined above is that of `Car3` which is connected to no wheel and the wheels `wheel9` .. `wheel12`connected to no car.

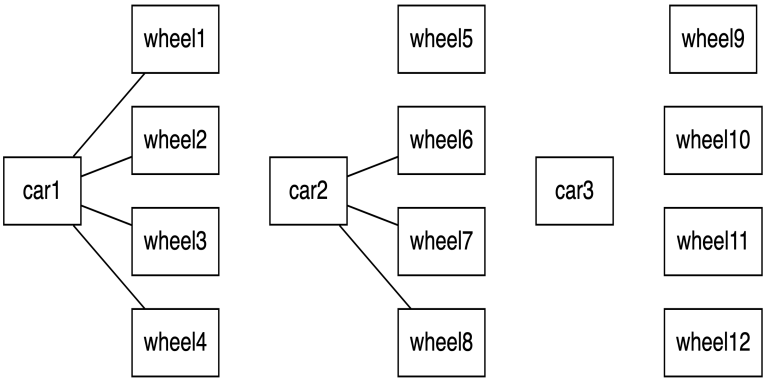


Figure 1.10.: Example object multiplicity

It is important to observe that the multiplicity notation of UML class diagrams is very different from what you have in ER, especially the notation in use at Politecnico di Torino.

The class diagram in Figure 1.9 can be described in UML as

Figure 1.11.: Example ER equivalent multiplicity

A special kind of association is **Aggregation**/composition, indicated by a diamond. It expresses a whole-part relationship; e.g., Car is composed of Engine, Wheel(×4), CDPlayer.

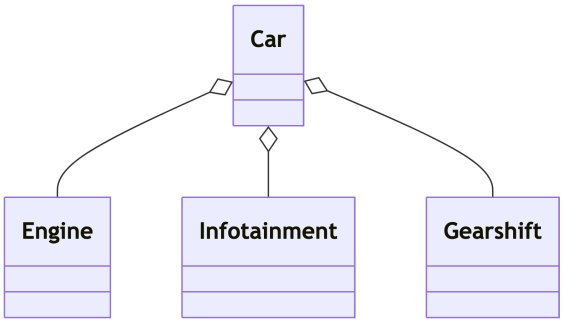
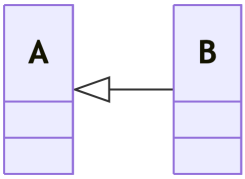


Figure 1.12.: Example of aggregation in UML

1.3.4. Inheritance

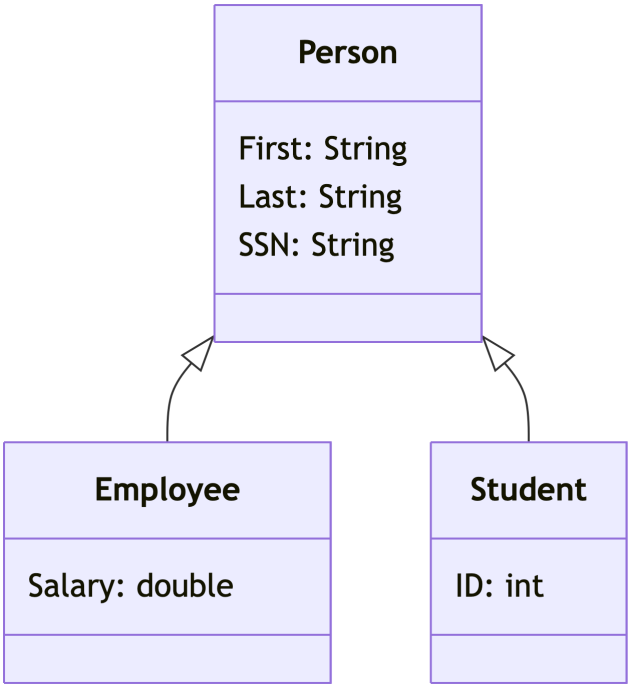
Inheritance A class can be a sub-type of another class. The inheriting class contains all the methods and fields of the class it inherited from plus any methods and fields it defines. The inheriting class can override the definition of existing methods by providing its own implementation. The code of the inheriting class consists only of the changes and additions to the base class.



B specializes A means that

- B has the same characteristics as A in terms of
  - Attributes
  - Methods
  - Participation in associations
- B may have additional characteristics

In practice B is a special case of A, or viceversa, A is a generalization of B.



Inheritance terminology

- Class one above: Parent class
- Class one below Child class
- Class one or more above: Superclass, Ancestor class, Base class

- Class one or more below: Subclass, Descendent class, Derived class

Motivation for using inheritance. Frequently, a class is merely a modification of another class. In this way, there is minimal repetition of the same code. Localization of code: fixing a bug in the base class automatically fixes it in the subclasses; adding functionality in the base class automatically adds it in the subclasses. Less chances of different (and inconsistent) implementations of the same operation Example of inheritance tree

Twitter (simplified) A registered user can Post a tweet Follow another user Reply to a tweet Add a like to a tweet

1.4. Building models

Essential guidelines:

- If a concept has significant properties and/or describes types of objects with an autonomous existence, it can be represented by a class.
- If a concept has a simple structure, and has no relevant properties associated with it, it is likely an attribute of a class.
- If a concept provides a logical link between two (or more) entities, it is convenient to represent it by means of an association.
- Any operation that implies access to the attributes of a class should be defined as a method.
- If one or more concepts are special cases of another concept, it is convenient to represent them by means of a generalization.
- When distinct classes may play the same role w.r.t. an association to a given class it is common to represent this commonality by generalization Inheritance includes also associations.

Modeling strategies:

- Top-down: Start with abstract concepts and perform successive refinements
- Bottom-up: Start with detailed concepts and proceed with integrating different pieces together
- Inside-out: Like bottom-up but beginning with most important concepts first
- Hybrid: a mix of the previous strategies

1.4.1. Model quality

- Correctness No requirement is misrepresented
- Completeness All requirements are represented
- Readability It is easy to read and understand
- Minimality There are no avoidable elements

2. Configuration Management with Git

Learning objectives

- Understand what is configuration management
  - What is Version Control
  - What are the main concepts of VC
- Learn how Git can be used for CM
  - local repository operations
  - remote repository operations
  - working with branches
  - conflict resolution
- Gitflow collaboration approach
- GitLab operations

2.1. Configuration Management

Software Configuration Management (SCM) is the discipline that applies technical and administrative direction and surveillance in order to(*IEEE Standard for Configuration Management in Systems and Software Engineering* 2012):

- identify and document the functional and physical characteristics of a configuration item,
- control changes to those characteristics,
- record and report change processing and implementation status, and
- verify compliance with specified requirements

The main issues faced by SCM are:

- What is the history of a document?
  This problem is solved with Versioning.
- Who can change what and how?
  This issue is addressed by Change control.
- What is the correct set of documents for a specific need?
  This issue concerns Configuration.

2.1.1. Versioning

The problem of managing successive versions of the same file (or more in general item) is quite common in the experience of everybody writing even the simplest document, and of course of any programmer.

The trivial technique of appending fragments to the name of a file does not work even in simple cases. Configuration management frames the problem in general terms and provides the instruments to solve it sistematically.

The generic tem *Configuration Item* (CI) indicates an aggregation of work products that is treated as a single entity in the configuration management process. A CI (typically a file):

- has an identifying name,
- all its versions are numbered and kept,
- it is possible to retrieve any previous version,
- the author decides to change version number with specific operation (*commit*).

While the former three charateristics are in common with versioning file systems (e.g. cloud file systems), the last one is peculiar with SCMs: it is the author of the CI that decides when to *commit* a new version, independently of saving a file.

Any SCM must provide a *Version numbering*, which is a simple naming scheme that in the simpleste case uses a linear derivation e.g. V1, V2, V3, etc. Ofen the derivation structure is a tree or a network rather than a linear sequence. Version names by themselves are not usually meaningful as long as they fit in sequencing scheme.

The term *Configuration* is used to indicate a set of CIs, each in a specific version, it is a snapshot of a software system at certain time. It includes several CIs, each in a specific version. The same CI version may appear in different configurations.

A *Baseline* is a configuration in stable, frozen form. Not all configurations are baselines, some are just temporary states of the system and are not meant to be preserved. The main types of baselines are:

- Development – for internal use
- Product – for delivery

2.1.1.1. Semantic Versioning

A commonly used approach to assign version numbers to products is Semantic Versioning. This approach is usually applied to a whole system – a baseline – and not to individual CIs.

With semantic versioning, product version numbering are based on a pattern that include three components:

MAJOR . MINOR . PATCH

The specific elements are incremented when specific changes occurr in the products:

- `MAJOR`: when you make large (possibly incompatible) API changes,
- `MINOR`: when you add functionality in a backwards-compatible manner, and
- `PATCH`: when you make backwards-compatible bug fixes.

Sometimes it is possible to add the tag `-SNAPSHOT` to the tail of the version, e.g. `1.0.0-SNAPSHOT`. The `SNAPSHOT` value refers to the *latest* code along a development branch, and provides no guarantee the code is stable or unchanging. Conversely, the code in a *release* version (any version value without the suffix `SNAPSHOT`) is unchanging.

In other words, a SNAPSHOT version is the *development* version before the final *release* version. As such, the SNAPSHOT is older than the corresponding release with the same Major.Minor.Path number.

During the release process, a version of `x.y-SNAPSHOT` changes to `x.y`. The release process also increments the development version to `x.(y+1)-SNAPSHOT`. For example, version `1.0-SNAPSHOT` is released as version `1.0`, and the new development version is version `1.1-SNAPSHOT`.

2.1.2. Change control

A *Repository* is a collection of all software-related artifacts (CIs) belonging to a system. The term is used also to indicate the location/format in which such a collection is stored.

The repository takes care of storing all the versions of the configuration items and to increment the version count according to the adopted scheme.

There are two fundamental oprations that can be perfomed on a repository:

- **Check-out** Extraction of CIs from the repository, into the local file system, typically with goal of either changing them or use them (e.g. for compiling).
- **Check-in** (or **Commit**)
  - Insertion of new versions of CIs in the repository.

Software development team members write code simultaneously and access CIs both reading and commiting new versions. Problems can arise when different developers modify the same CI concurrently, resulting into a conflict.

The most simple implementation of a repository is a common repository (e.g., like a shared folder), where everybody can read/write files, concurrently. The working of a *common repository* in case of concurrent modification is described in Figure 2.1.

- Pro:
  - Very simple to implement and use
- Cons:
  - Conflicts are very frequent
  - Parallel work is possible but only on distinct CIs

To avoid conflicts and the ensuing risk of losing part of the work, a possible approach is to adopt the *lock-modify-unlock* (or serialization) strategy. Only one developer at a time can change a given CI. The behavior of such a strategy is described in Figure 2.3.

- Pro:
  - Conflicts are impossible

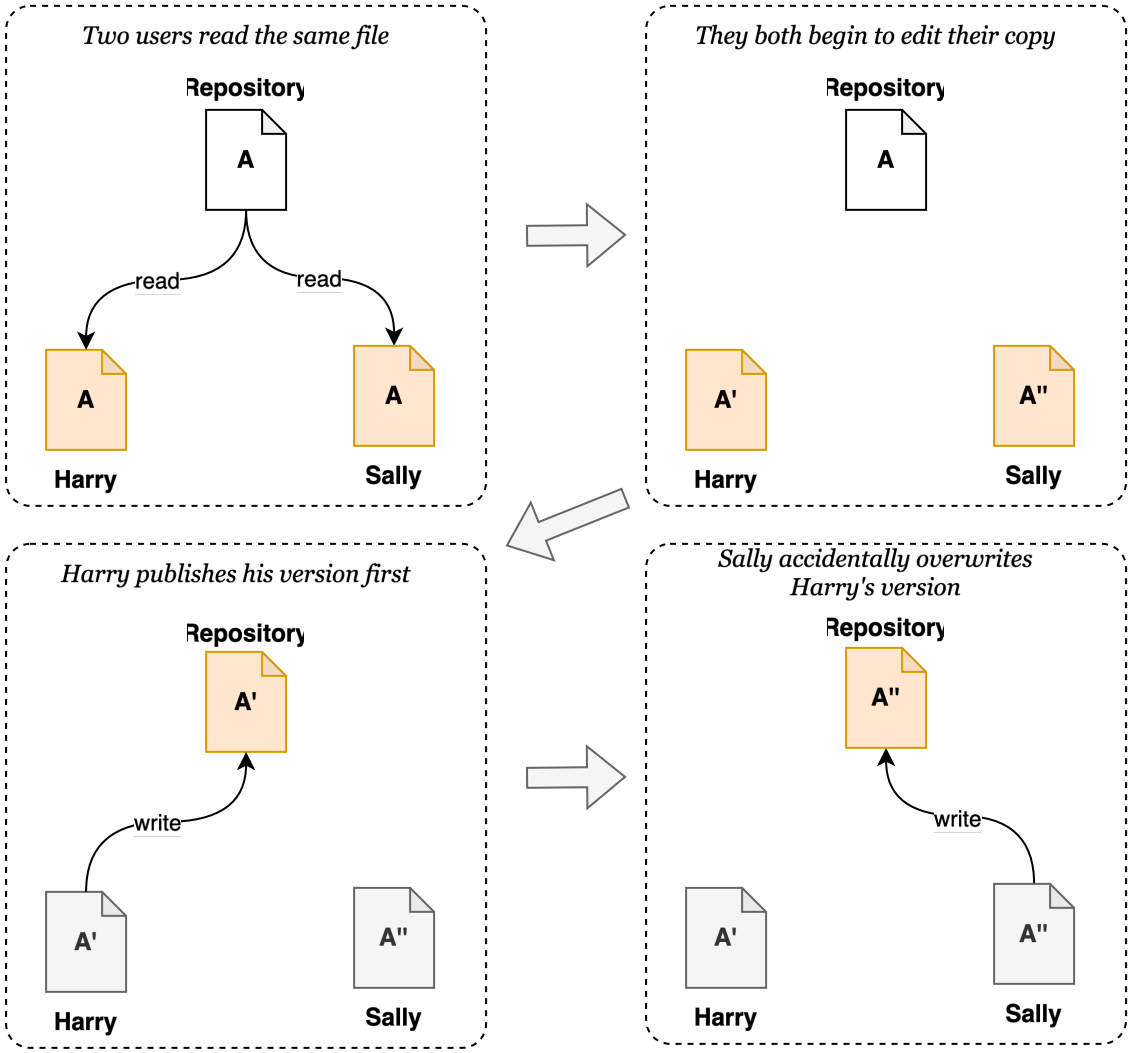


Figure 2.1.: Change control on file system

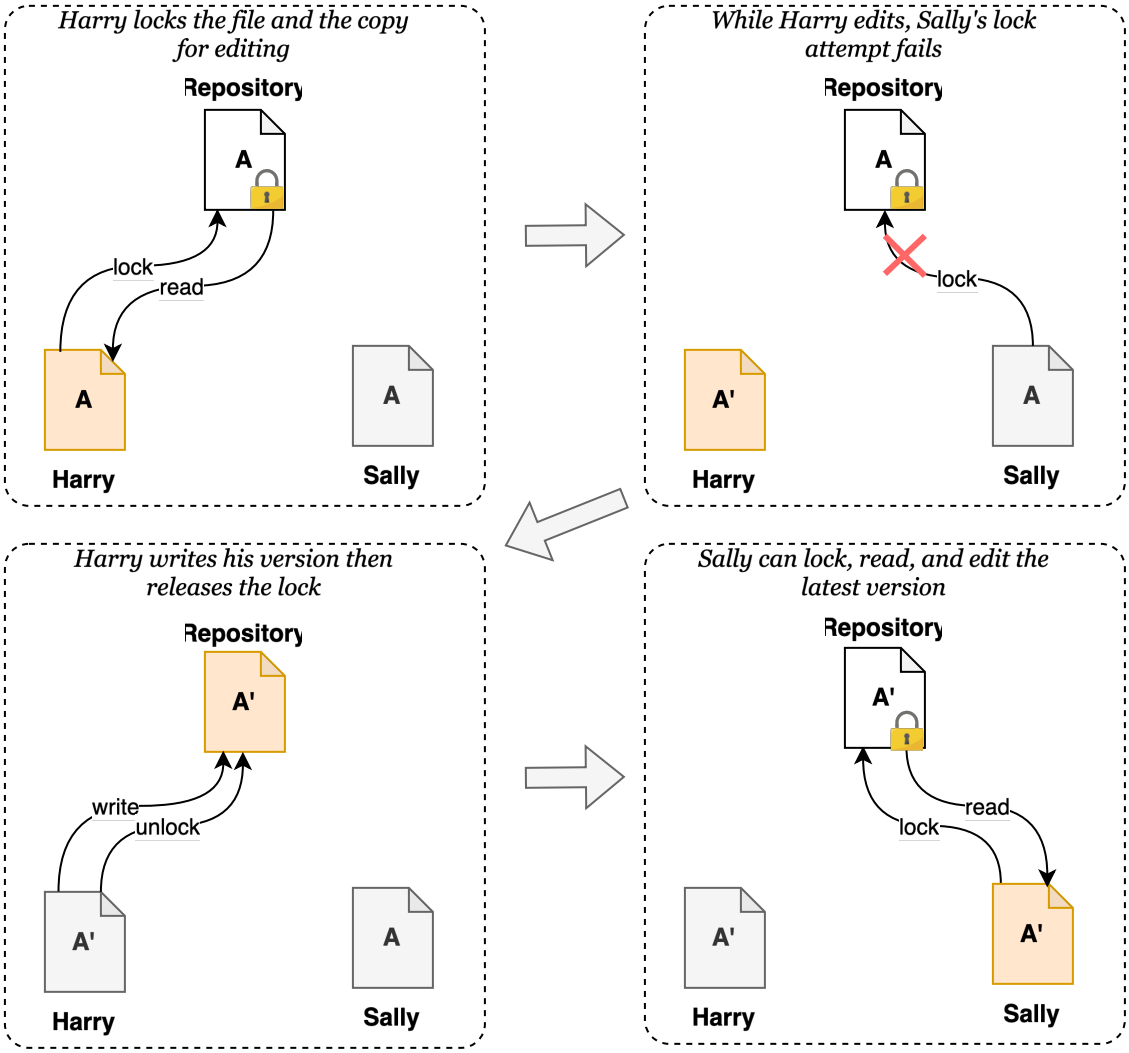


Figure 2.2.: Change control with lock-modify-unlock

- Cons:
  - No parallel work is possible, large delays can be introduce in case of non-responsive developers
  - Developers can possibly forget to unlock so blocking the whole team

To solve the limitations of the above model, it si possible to adopt the *copy-modify-merge* approach. It is possible to make many change in parallel to the same CI, when such concurrent changes occurr, they are detected and a *merge* is required to reconcile them.

- Pros
  - More flexible
  - Several developers can work in parallel
  - No developer can block others
- Con:
  - Requires care to resolve the conflicts

Most modern CSMs work using the copy-modify-merge approach.

2.1.3. Configuration

A *Branch* is a line of development that exists independently of other lines, yet still shares a common history when looking far enough back in time. A branch always takes life as a copy of some other branch, and moves on from there, generating its own independent history.

Branches may represent different configurations, e.g.,

- by platform,
- by milestone,
- by features.

Branches allow working in isolation form the other branches, especially the main branch where the released product configuration resides. Many new features or fixes can be developed independently and concurrently, each in its own relative branch. When work is complete in a branch, it can be verified and then merged into the main branch.

2.1.4. Repository Architecture

Architectures can be classified along two main axes:

- distribution, and
- data management.

Distribution concerns where repository are located:

- Local A simple database that kept all the changes to files under control.
  Example product:
  - RCS

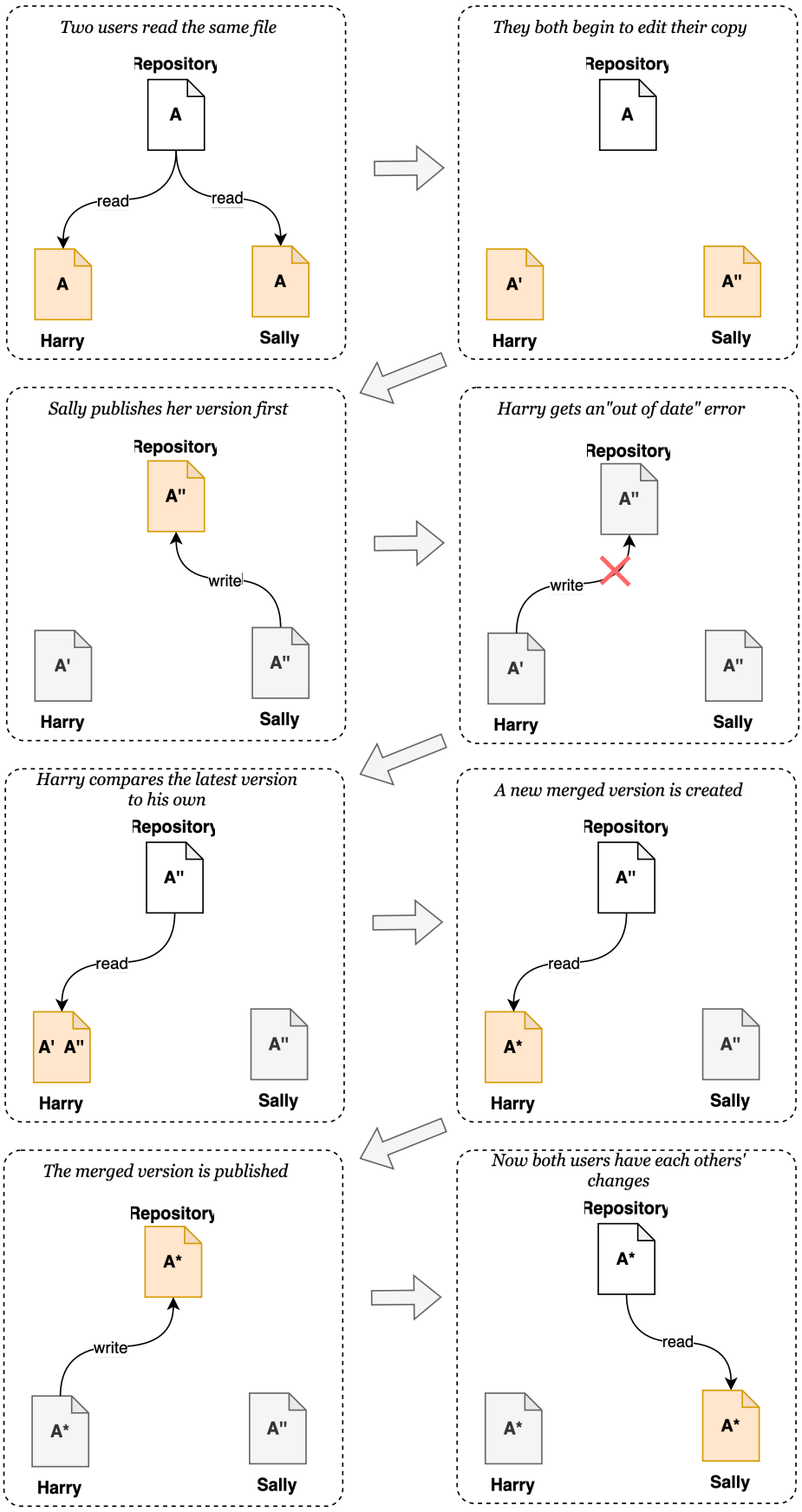


Figure 2.3.: Change control with copy-modify-merge

- Centralized A single server that contains all the versioned files, and several clients that check out files from that central place,
  Example products:
  - CVS
  - SCCS
  - PCVS
  - Subversion
- Distributed Clients fully mirror the repository locally and then perform synchronization.
  Example products:
  - BitKeeper
  - Mercurial
  - Fossil
  - Git

There are two main data Management Models

- Differences: information is kept as a set of files and all the changes made to each file over time, see Figure 2.4.
  - Pros: the size of the repository is small
  - Cons: to reconstruct the status of a file at a given time all changes must be re-applied in order
- Snapshots: every commit, a picture of what all your files look like at that moment is taken and the system stores a reference to that snapshot. If files have not changed, system stores just a link to the previous identical file, see Figure 2.5.
  - Pros: the status of a file at a given time is immediately available
  - Cons: the size of the repository is large

2.1.5. Most relevant SCM systems

Comparison table summarizing the **Version Control Systems (VCS)** you listed — including their **year of introduction**, **architecture type**, and **data management approach**:

VCS	Year Introduced	Architecture Type	Data Management Approach	Notes
SCCS (Source Code Control System)	1972	Local	Diff-based	First known VCS; stored file deltas line-by-line.
RCS (Revision Control System)	1982	Local	Diff-based	Improved SCCS; reverse deltas for faster access to latest version.
CVS (Concurrent Versions System)	1986	Centralized	Diff-based	Built on RCS; allowed multiple developers and network access.

VCS	Year Introduced	Architecture Type	Data Management Approach	Notes
PCVS (Personal Concurrent Versions System)	~1990s	Centralized / Local	Diff-based	Adapted CVS for single-user or small-team workflows.
Subversion (SVN)	2000	Centralized	Diff-based	Designed as "better CVS"; supports atomic commits and directories.
BitKeeper	1998	Distributed	Changeset (diff-based)	Early DVCS; influenced Git's design; used for Linux kernel development.
Mercurial	2005	Distributed	Snapshot-based	Similar to Git but simpler; each commit stores a full snapshot with compression.
Fossil	2006	Distributed	Snapshot-based	Self-contained system with integrated wiki and issue tracking.
Git	2005	Distributed	Snapshot-based	Uses content-addressable snapshots (SHA-1/2); dominant modern VCS.

2.2. Version Control with Git

Key characteristics of Git:

- Distributed SCM
- Uses snapshots
- Exploits local operations
- Has integrity checking: everything is check-summed before it is stored and can be reference by that checksum

There are many code hosting web applications and services built on top of git:

- GitHub
  - Widely used by OSS projects
  - Commercial use is growing
- GitLab
  - Largely used
  - Based on the open-source GitLab software
- BitBucket
  - Commercial
- Codeberg
  - open
  - uses the Forgejo open-source software

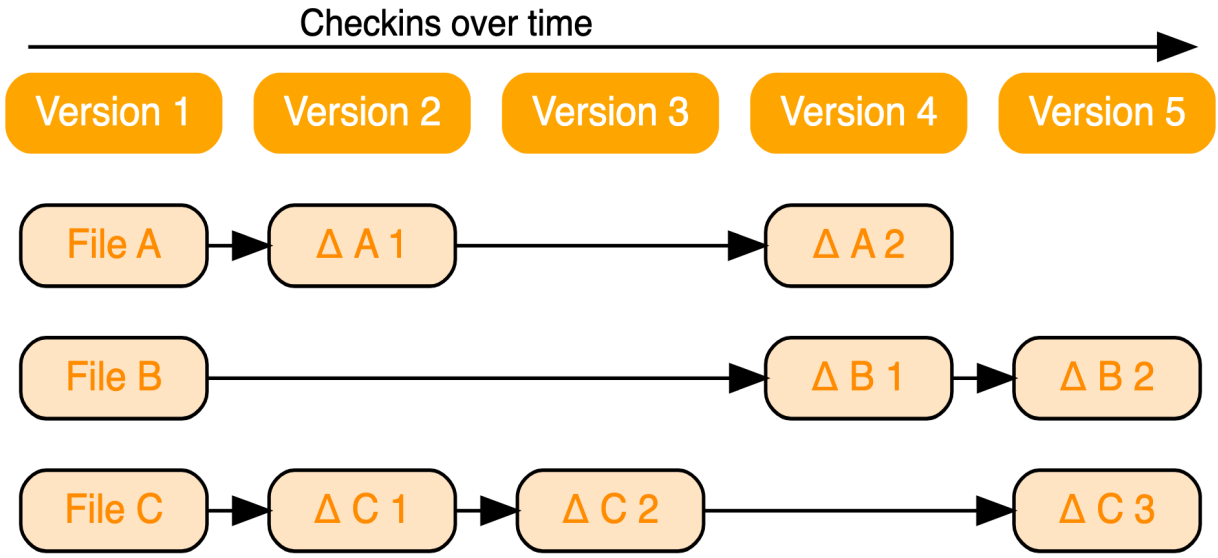


Figure 2.4.: Difference based versioning

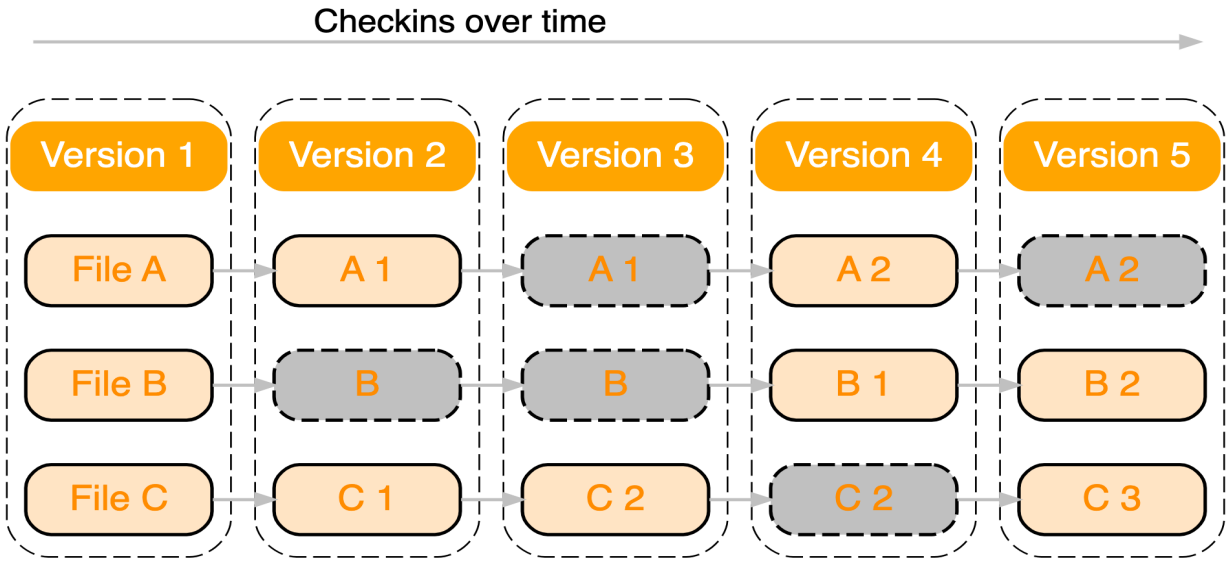


Figure 2.5.: Snapshot based versioning

2.2.1. Key concepts

Repository: place where you store all your work It contains every version of your work that has ever existed: - files - directories layout - history

`It can be shared by the whole team`

Commit: an operation that modifies the repository storing a new version of files: - It is atomically performed - The integrity of the repository is assured - It is mandatory to provide a log message (or comment) to explain the changes made, the message becomes part of the history of the repository

Every commit must be accompanied by a log message (if not provided an editor is automatically opened to enter one).

Conventional Commits is a lightweight convention on top of commit messages. Provides an easy set of rules for creating an explicit commit history. The typical structure of the log messages is:

```
<type>(<scope>): <subject>
<body>
<footer>
```

The common change types are

- `fix`: a commit of the type fix patches a bug in your codebase. It is typically linked to a PATCH increment in Semantic Versioning. E.g., `fix(middleware): ensure Range headers adhere more closely to RFC 2616`
- `feat`: a commit of the type feat introduces a new feature to the codebase It is typically linked to a MINOR increment in Semantic Versioning. E.g,

  `feat: allow provided config object to extend other configs`

- `BREAKING CHANGE`: a commit that has a footer BREAKING CHANGE:, or appends a ¡ after the type/scope, introduces a breaking API change. It is typically linked to a MAJOR increment in Semantic Versioning. E.g.,

  ```
  feat!: send an email to the customer when a product is shipped
  BREAKING CHANGE: `extends` key in config file is now used for extending other config files
  ```

The *body* part is a detailed list of the changes that are included in the commit.

The *footer* part is used to link the commit to relevant items. A typical usage is to mention the issue number that the changes in the commit address.

Example:

```
fix(middleware): ensure Range headers adhere more closely to RFC 2616 <1>
Added one new dependency, use `range-parser` (Express dependency) to compute range. <2>
It is more well-tested in the wild.                                                 <2>
Fixes #2310                                                                  <3>
```

1. the change is a `fix` and it affects the `middleware` component, the subject describes in short the change;
2. the body describes in more detail the change;
3. the footer mentions the issue `#2310` in the issue-tracking system.

Working Copy: it is a snapshot of the repository where the users can make changes. It is private for a single developer, not shared by the team. It also contains some metadata so that git can keep track of the state of files.

The possible categories of the files in the Git Working Copy are:

- *Tracked*: files in the working copy (file system) that are monitored by Git
- *Untracked*: files present in the working copy but whose change are not monitored
- *Ignored*: files present in the working copy completely ignored by Git Ignoring Files is possible to exclude permanently from version control some files in the project folder. These files must be listed in the `.gitignore` file so that such files or folders will not be considered by Git.

Staging Area: a sort of temporary stocking area. It contains a snapshot of the files that are have been selected to be included in the next commit. Also called "index".

The possible status of any tracked file in the *Working Copy* is:

- *Not changed*: in the original state as in the latest reference commit
- *Modified*: changed only in the local working copy (file system)
- *Added*: just added to staged files without any previous version in the repo
- *Staged*: modified and marked in its current version to be included in the next commit (i.e. included in the Staging Area)

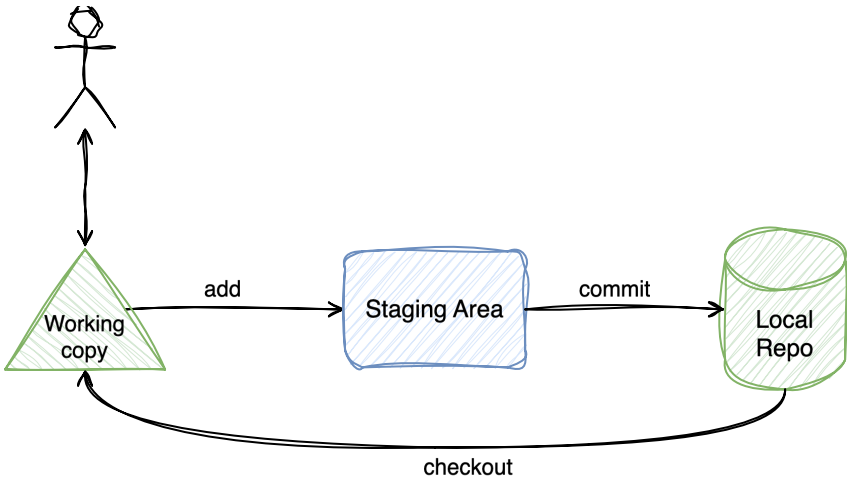


Figure 2.7.: Local working elements

The typical local workflow is:

- Modify or create files in your working copy, typically using you favorite IDE,
- Stage the files, adding snapshots of them to your staging area

  ```
  git add /path/to/file
  ```

Note the same command adds a previously untracked file or a modified file to the staging area

- Commit them, Git takes the files as they are in the staging area and stores that snapshot permanently to your (local) repository.

  ```
  git commit --message "feat: requirement R1 implemented"
  ```

2.2.2. Interact with remote repository

The local repository is typically used by a single developer on their machine. While a remote (shared) repository is used to keep in sync several local repositories and allow combining the independent work from different developers. The remote repository is often called `origin`.

There are two main operations to allow synchronizing the local repository with the remote one:

- **push**: pushes (i.e. copies) changesets (set of commits) from a local repository to a remote (shared) one.

  ```
  git push
  ```

- **pull**: updates the local working copy with the latest contents of the remote repository by pulling the remote changesets.

  ```
  git pull
  ```

  The pull operations is actually a shortcut for two separate operations:

  - **fetch**: retrives the recent changesets from the remote repository and save them into the local repository
  - **checkout**: copies the latest version of the files from the repository into the working copy

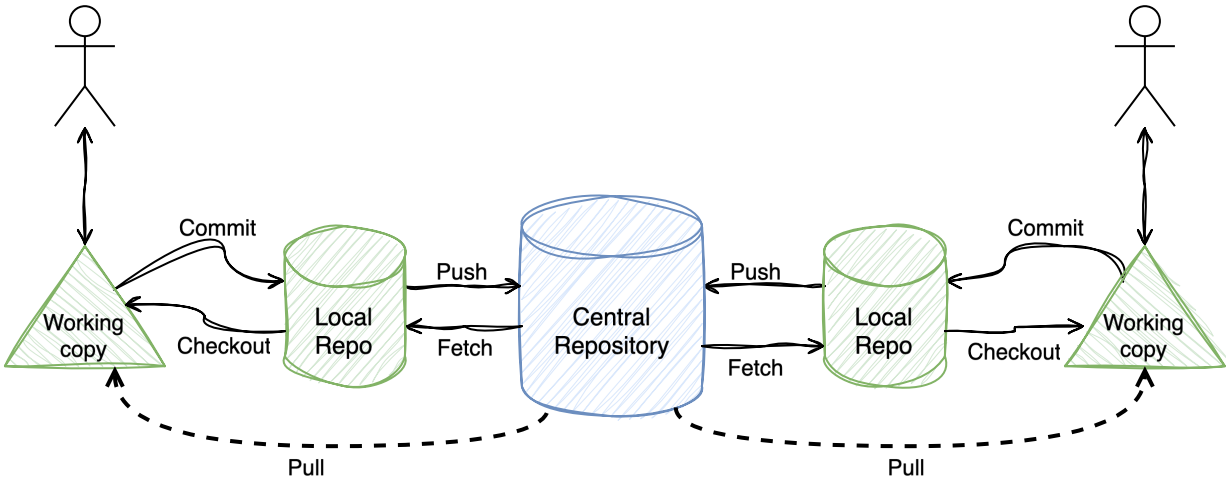


Figure 2.8.: Key elements in git collaboratio

Commit graph: a directed graph maintained by git where nodes are commits and edges link commits to their immediate predessors in history.

Git keeps track of the latest commits that have been synchronized with the remote repository. When a synchronization between repositories is performed (i.e., push, pull, fetch) the changed portion of the graph is exchanged.

`HEAD`: a pointer to the lastes commit that is the one that is currently checked-out in the working copy. It is the reference to detect changes that can be staged and committed.

When a new commit is created, its predecessor is set to the current `HEAD` and `HEAD` points to the latest commit.

Usually `HEAD` should coincide with the current branch in order to create a linear history

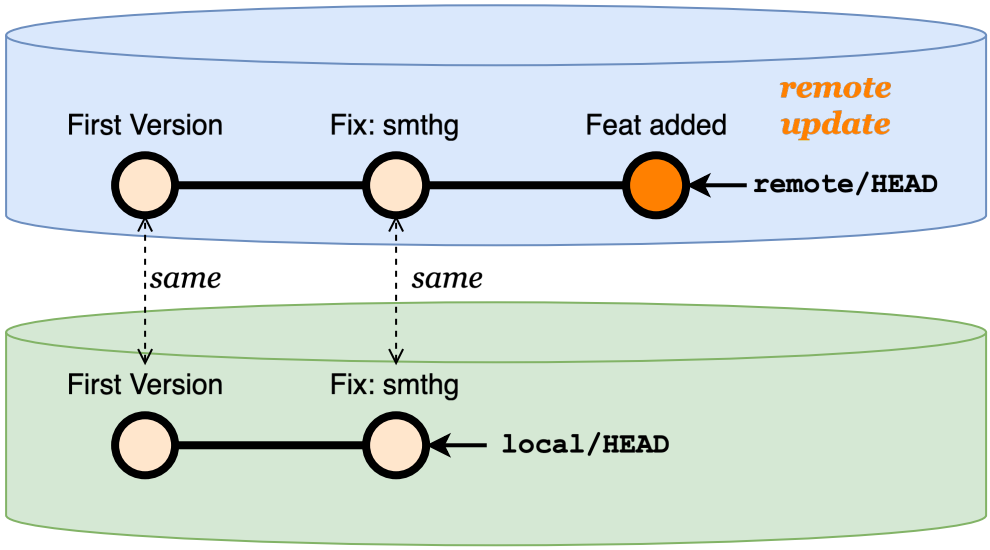


Figure 2.9.: Remote updates

The role of the central (remote) repository is to collect updates from all developers.

- `git fetch` to dowload locally the remote commits, but without altering `HEAD` or touching the working area

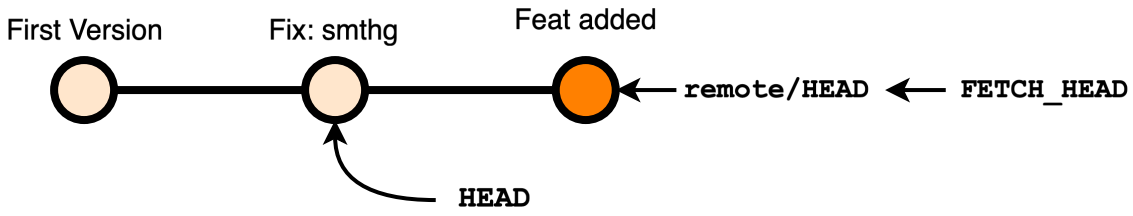


Figure 2.10.: Fetching remote updates

- `git merge -ff` merge the remote commit into the local branch moving `HEAD` to the latest commit from the remote and checking out the lastes changes into the working area

The two operations are usually performed in a single step using the `git pull` command.

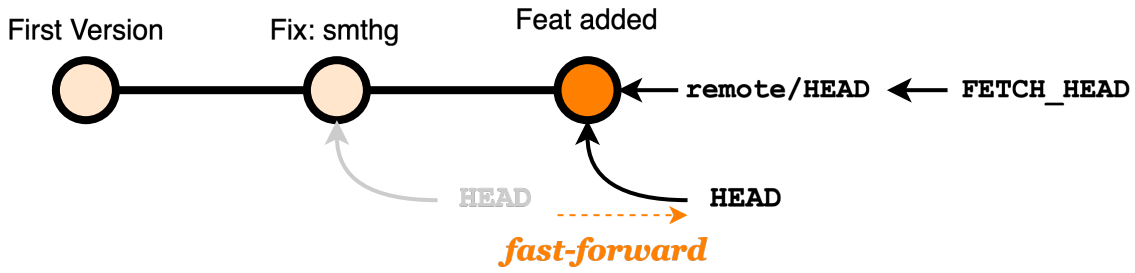


Figure 2.11.: Mergin remote updates into current branch

2.2.3. Diverging commits

Diverging commits are two (or more) distinct commits that share the same predecessor in the same branch. A divergence usually happen when two developers work concurrently on the same branch. One pushes the new commit to the remote and the other has performed a local commit.

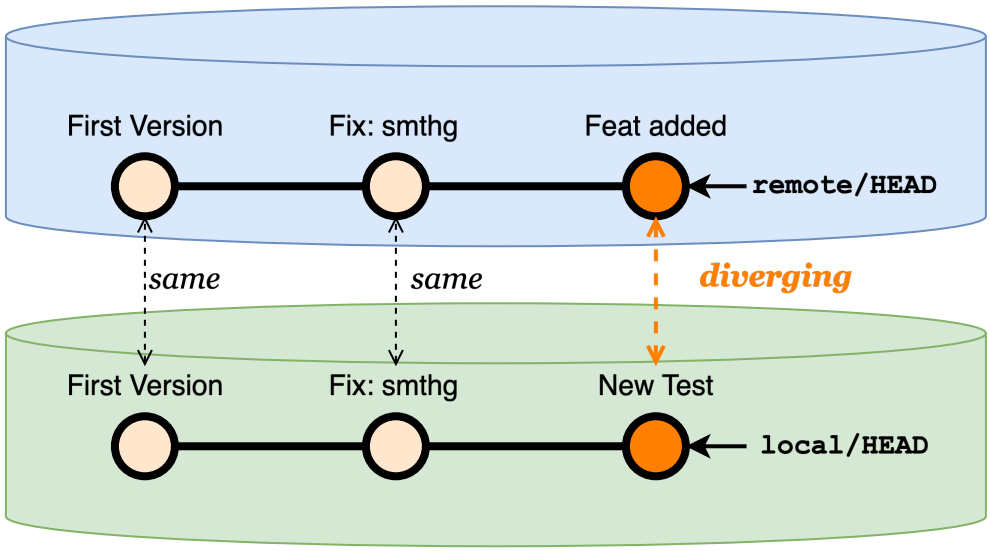


Figure 2.12.: Remote concurrent updates

When the remote updates are fetched locally the two commits form a divergence:

In presence of concurrent commits Push fails because there are new changes (commits) in the remote repository and it is impossible to decide:

- which of the new commits should become the new HEAD
- whether commits should be reordered (and how) or combined

Also the Pull command fails because a checkout is not possible due to the divergent commits. Only the Fetch part of Pull succeeds. The resulting local commit graph is divergent:

There are two possible strategy to resolve the ambiguity deriving from diverging commits:

- **rebase**
- **merge**

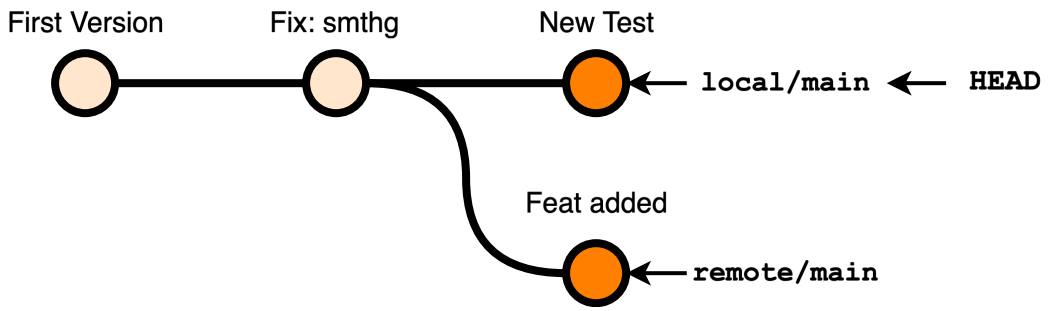


Figure 2.13.: Diverging commits

The rebase approach rebases the local `HEAD` on top of the remote `HEAD`

- `git pull --rebase` # fetch + rebase
- `git rebase` # only rebase

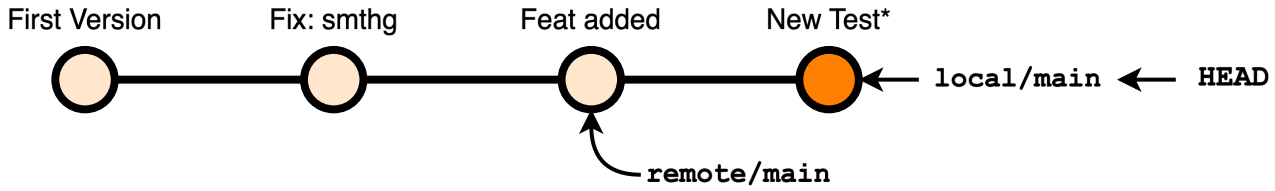


Figure 2.14.: Rebased diverging commits

The merge strategy consists in creating a new *merge* commit that combines the changes found in the two diverging commits:

-`git pull --no-rebase` # fetch+merge

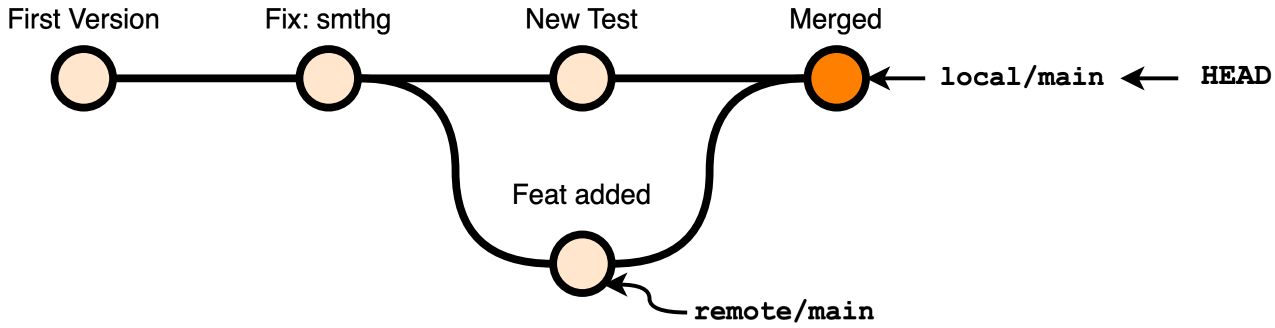


Figure 2.15.: Merged diverging commits

2.2.4. Git Branches

A branch is simply a lightweight movable pointer to a commit. The default branch name is `main`[1].

[1]The default branch used to be called `master` until a few years ago. The developers' community felt that the term *master* had negative connotation therefore in 2020 it was changed into `main`. Keep in maind that you could find older repositories still using that name.

The current branch is the pointed to by `HEAD` and it moves forward when a commit is performed in the current branch.

Usually branches are created starting from the current branch:

- `git branch testing`

  Creates a branch named *testing*

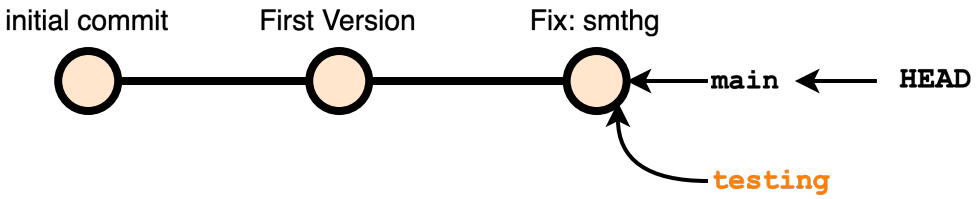


Figure 2.16.: Creating a branch

  Remember: the `branch` command does not switch to the new branch, to both create and switch to the branch it is possible to use `git swithc -c testing` the `-c` option creates the branch if not already present.

- `git checkout testing` or `git switch testing`

  Moves the `HEAD` pointer to the *testing* branch and possibly updates the files in the working copy (if the branche contains any changes)

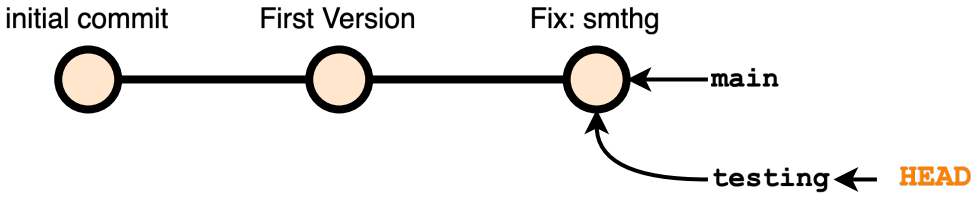


Figure 2.17.: Switching to branch / Checking out branch

Branches are used to perform work in isolation. When the work in a branch has been completed it must be carried on back into the original branch and integrated with other possible changes occurred in the meantime.

There are three possible strategies:

- Fast-Forward Merge
- Merge
- Rebase

Fast-Forward Merge is possible only if no additional commits have been added to the `main` branch since the working branch has been created.

```
git checkout main                                                    ①
git merge testing                                                    ②
```

① The current branch must be `main`

② Performs a fast-forward merge

The operation is illustrated in the following Figure 2.18.

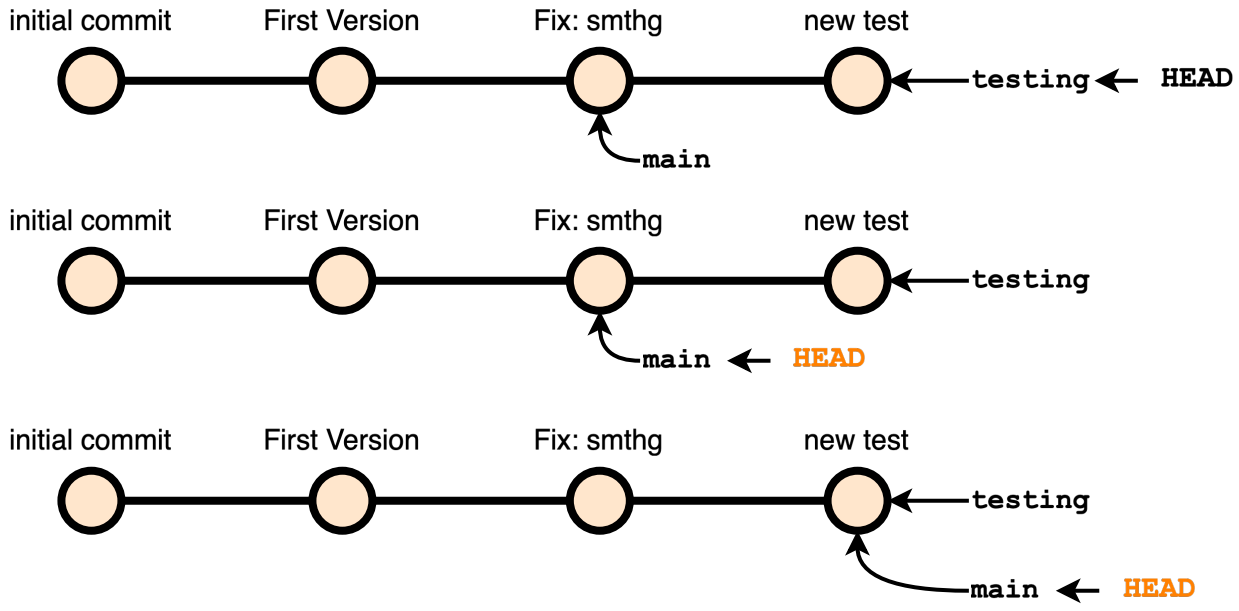


Figure 2.18.: Fast-forward Merge

Three-way merge. The second option, merge into `main` can be used in any case when there are new commits in the `main` after the working branch was created.

```
git checkout main                                                    ①
git merge testing                                                    ②
```

① The current branch must be `main`
② Merge from `testing` into `main` (= `HEAD`). Performs a three-way merge, i.e. it combines the two branches with the common ancestor

Rebase. The third option, rebase onto main and then fast-forward, can be used in any case, like the merge.

```
git rebase main                                                      ①
git checkout main                                                    ②
git merge testing                                                    ③
```

① Rebasing `testing` (= `HEAD`) onto `main`. The commits in `testing` are modified to incorporate the changes added to `main` after the branching. The branching point is moved to the latest commit in `main`.
② Switching to `main`
③ Perfomrm a fast-forward (like above)

The operations are illustrated in Figure 2.20

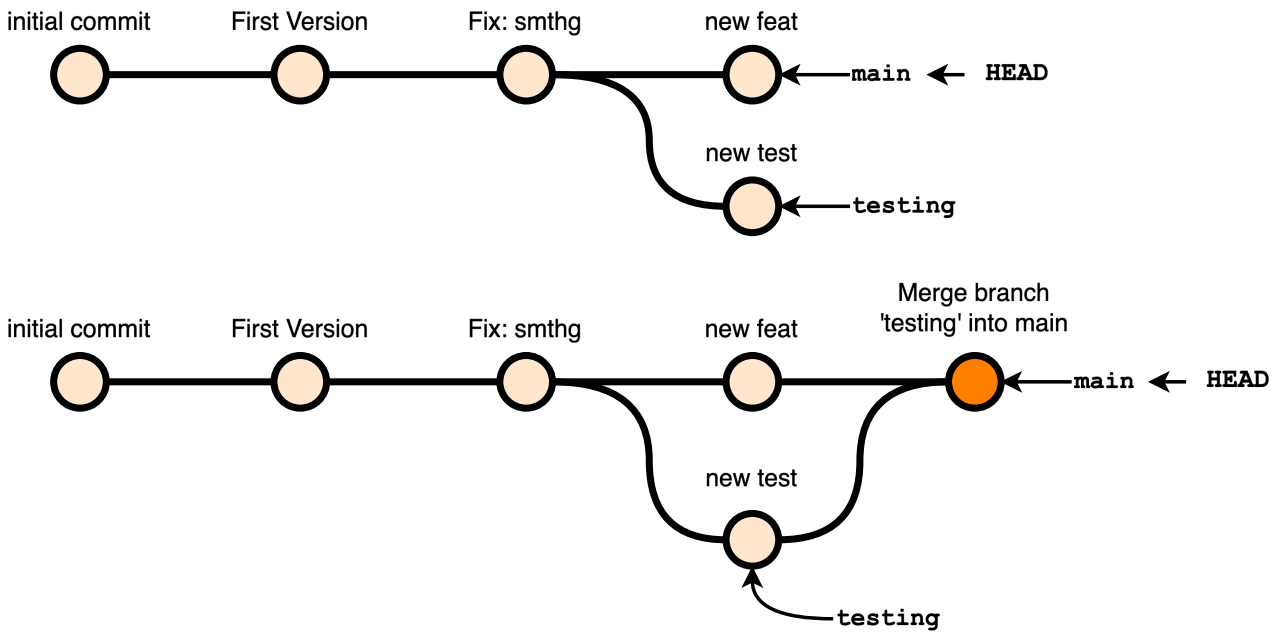


Figure 2.19.: Three-way Merge

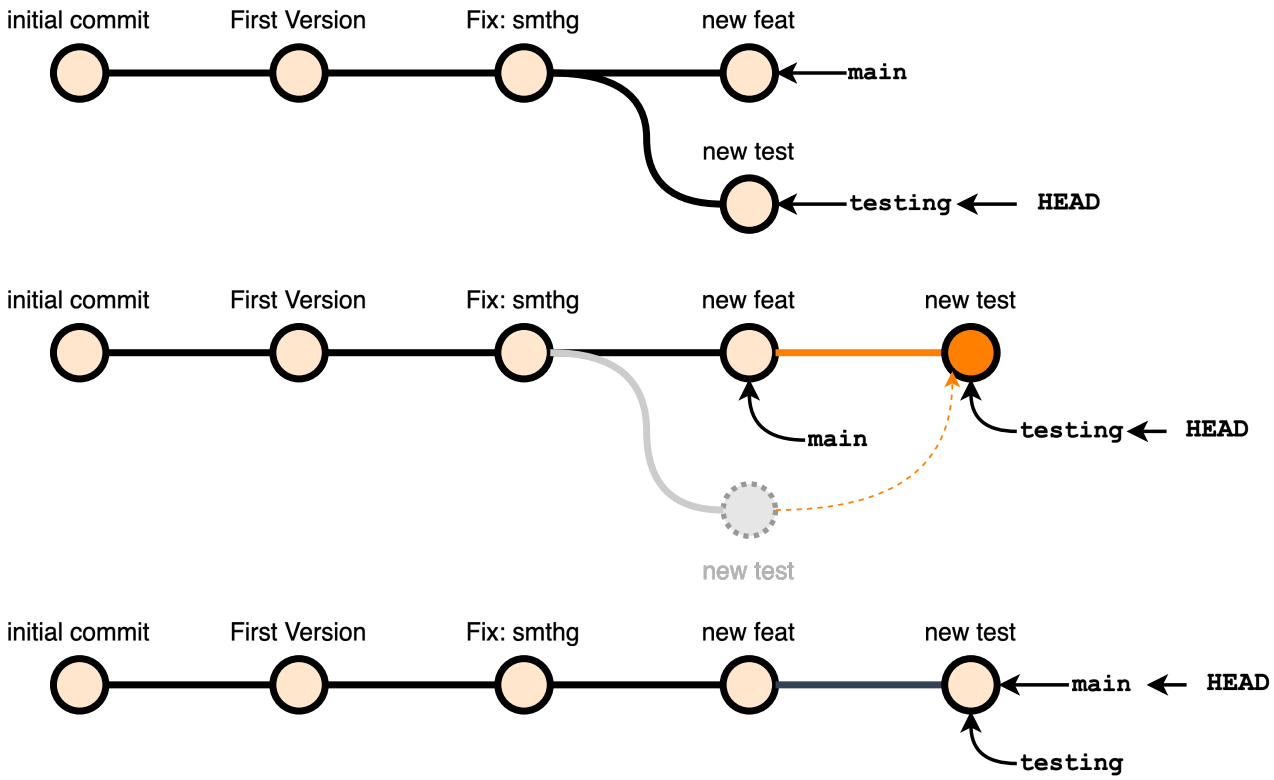


Figure 2.20.: Rebasing branch on main and fast-forward

Comparing the final commit graphs resulting from merge (Figure 2.19) vs. rebase (Figure 2.20), the difference between the two operations is clear:

- merge preserves all the previous commits and results into an history that forks and then joins;
- rebase alter the commits to adapt the the new base and linearize the history.

2.2.5. Synchronizaton between branches

It can happen that some changes committed to a branch (e.g., `main`) should be incorporated also in another branch but the work in the branch is not ready yet.

A typical example is when a critical bug fix has been merged into `main`, independently from a longer work that is performed in a separate branch. The fixed code should be integrated in the work in progress in order to have a complete and up-to-date code to work on.

In this case it is possible to

- rebase the working branch onto `main`, like you would do to include the changes to `main` but without performing any fast-forward, this is shown in Figure 2.21;

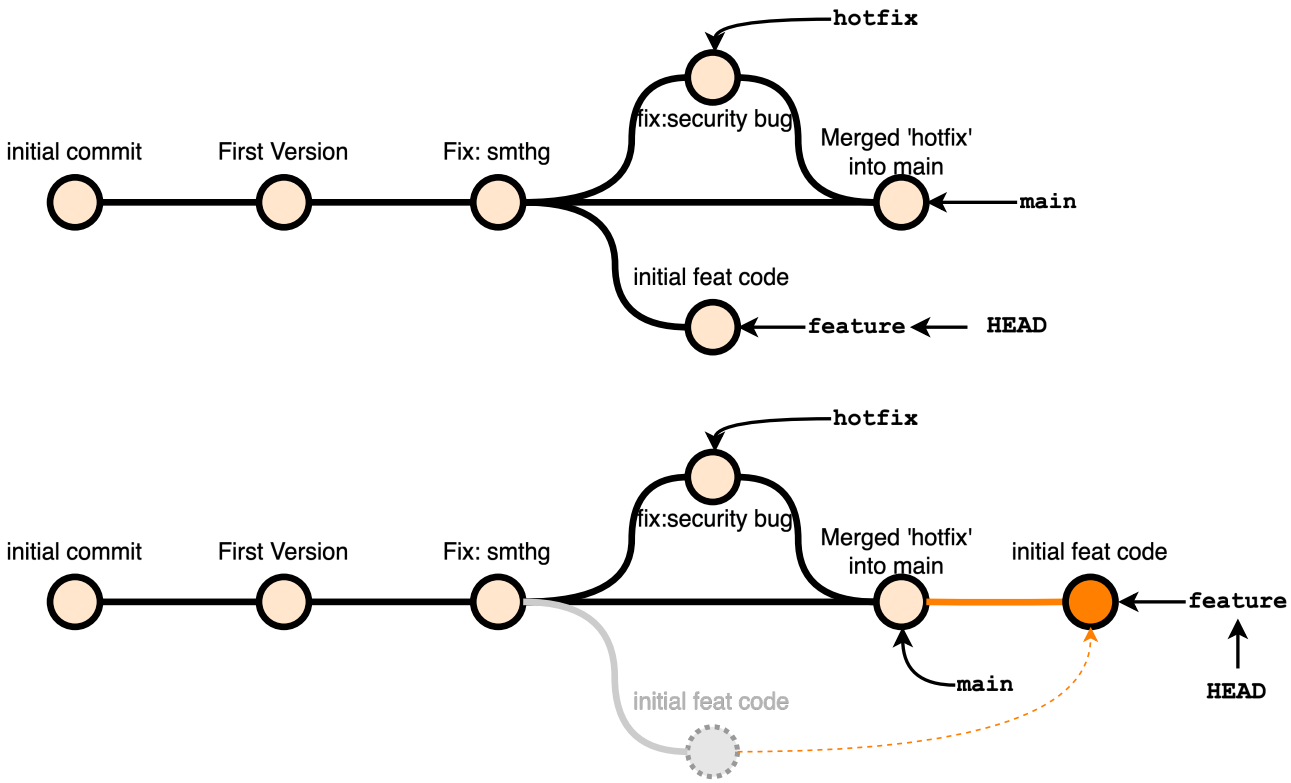


Figure 2.21.: Rebasing a branch on main to incorporate fix

- merge the `main` branch into the working branch (the opposite of what done previously), this is shown in Figure 2.22.

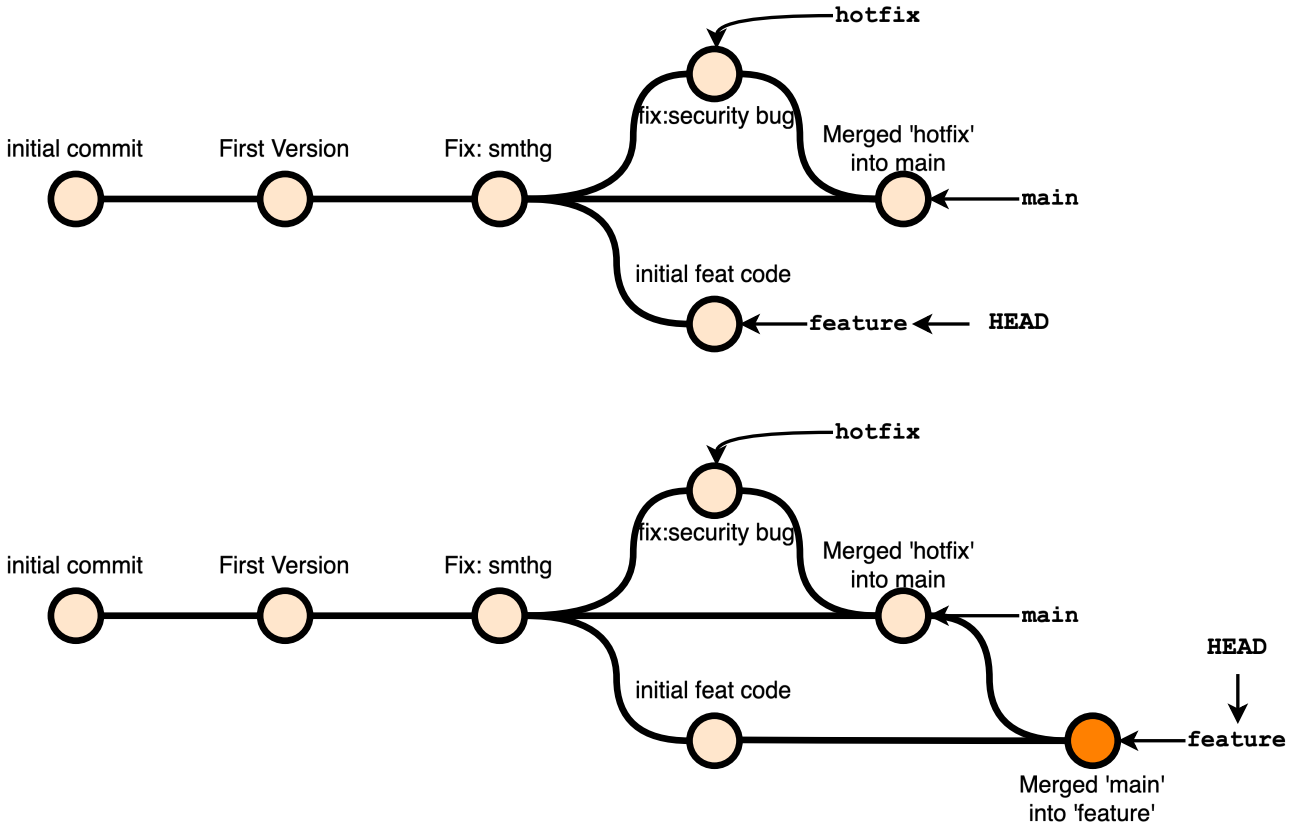


Figure 2.22.: Merging a fix on main into a branch

Either way, after the working branch has been updated with the lastest changes in the `main` branch, the work can continue as usual. Eventually the branch will be merged into or rebased onto `main` when the work is complete.

2.2.6. Conflicts

When a merge commit is created – or when previous commits are altered to perform a rebase –, git automatically combines the changes from the two branches at its best. Typically, when multiple changes are applied concurrently at different places in the file, no conflict is detected and the changes are automatically merged.

An example is shown in Figure 2.23. The `main` branch contains the changes to Section 1, while branch `ch-2` includes the changes to Section 2. A merge can be easily perfomed combining both changes into a single file.

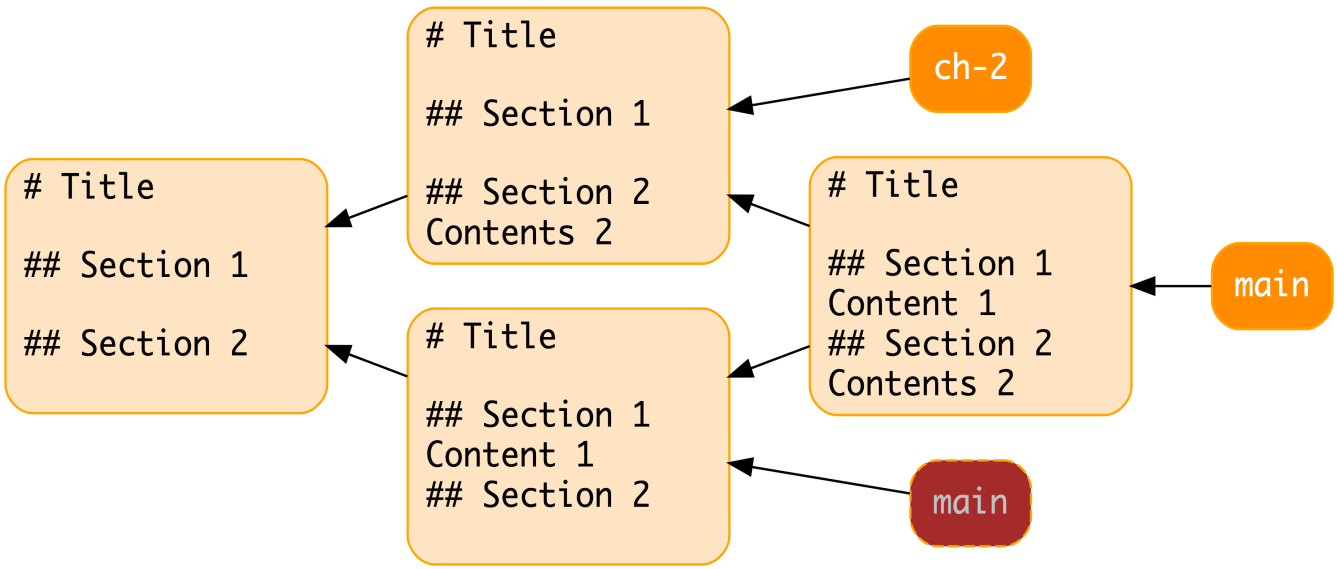


Figure 2.23.: Example of two branches with NON-conflicting changes.

A conflict arises, upon a `merge` or a `rebase` (possibly triggered by a `pull`), when one or more files have undergone changes that cannot be reconciled automatically. Typically overlapping ranges of lines have been modified by two distinct diverging commits which can belong the same branch or distinct branches.

An example of changes that cannot be reconciled automatically is show in Figure 2.24. While the top and bottom lines are kep in both branches, the filling is different in the two branches. The conflicts are highlighted within the files.

The merge or rebase operation is suspended until conflicts are resolved.

The command `git merge blt` that attempts to perform an automatic merge, fails. The working area then contains a copy of the conflicting file that contains all conflicts:

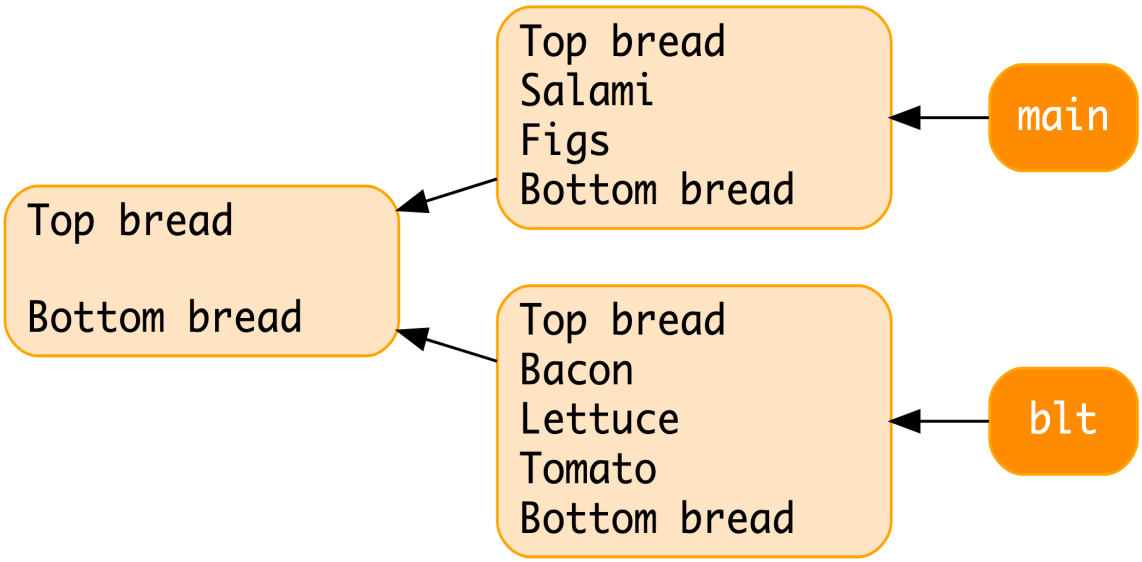


Figure 2.24.: Example of two branches with conflicting changes.

```
Top bread
<<<<<<< HEAD
Salami
Figs
=======
Bacon
Lettuce
Tomato
>>>>>>> blt
Bottom bread
```

The conflicts must be manually resolved by editing the files, combining the conflicting changes and eventually removing tall the conflict markers (`<<<<<<<` and `>>>>>>>`). Often and IDE provide a combined view of the two versions and the result to easily resolve the conflicts.

After the conflicts are solved, the conflict-free files must be staged with `git add`.

And finally the reconciled changes can be commtted:

```
git commit --message "merged blt"
```

If the conflict emerged during a rebase opreation, the conflict resolution withing the file is performed in the same way, the only difference is that the rebase operation must be resumed:

```
git rebase --continue --message "rebased"
```

Rebase vs. Merge:

- Rebase

  - creates a linear commit history rewrites the commit history of the rebased branch
  - better suited for keeping a cleaner commit history.

- Merge
  - combines the commit histories of two branches/threads,
  - creates a new merge commit
  - better suited for integrating changes from different branches

Merge is generally considered safer than Rebase because it preserves the original commit history and creates a new merge commit, which makes it easier to undo the merge if necessary and understand what happened.

Graph structure: every merge commit increases the connectivity of the commit graph by one. A rebase, by contrast, does not change the connectivity and leads to a more linear history

Recommended personal work procedure

- Perform a Pull before you start coding
- Create a branch for your work and switch to it
- Merge with the main when complete
  - In agreement with other developers
- After merge you can potentially remove old unused branches

2.3. Team Collaboration with Git

The operations described in the previous section can be used in different ways to let a team of developers collaborate on a project.

2.3.1. Gitflow

GitFlow is a branching model designed to structure and streamline software development. It defines a clear workflow for managing code versions, ensuring stability, and facilitating collaboration in a team. Introduced by Driessen (2010), it is widely used in a large number of software development projects.

GitFlow is a model, not a strict rulebook. The core concepts remain the same, but companies adapt it. A few common variations are:

- Classic GitFlow (with release branches)
- Simplified GitFlow (no release branches)
- Trunk-Based Development (no dev, everything on main)

The selection of the variant depends on your team's workflow and deployment strategy.

The advantages of using GitFlow are:

- it separates stable code from ongoing development, reducing risks
- it allows multiple features to be developed in parallel on isolated branches, increasing efficiency
- it ensures quality and stability by integrating testing before release

- it facilitates collaboration using Merge Requests for code review
- it keeps track of releases using tags and structured versioning

It is especially beneficial for projects with periodic releases (e.g., Agile methodologies). Essential for teams working in CI/CD environments, where automated builds and deployments rely on a structured branching model.

In this section we will focus on the trunk-based development variant.

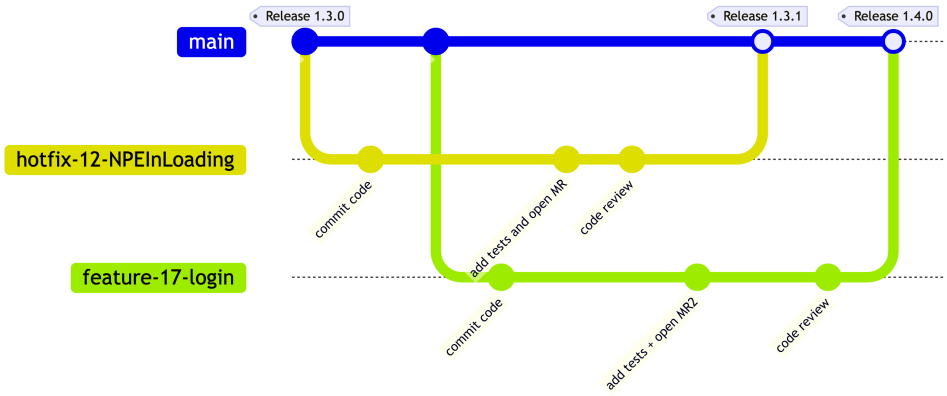


Figure 2.25.: Example of multiplicity in UML

Trunk-Based Development is a simple and widely used workflow in modern software development. It is based on having a single persistent branch (`main`) which always contains production-ready code. All new development happens in ephemeral short-lived branches, created for individual features or fixes. Ephemeral branches are regularly merged back into the main branch, after testing and code reviewing.

The main consequences of this approach are:

- the `main` branch is always stable and ready to release,
- the `main` consists of a clear, simple, and linear history,
- long-lived or multiple persistent branches are avoided, thus reducing complexity,
- it is posible to work on each feature or fix in an isolated temporary branch,
- allows parallel development on multiple features without interference,
- encourages frequent integration to reduce merge conflicts,
- enable collaboration through Merge Requests and Code Reviews.

Trunk-based developments uses different type of branches:

- Persistent branches: These branches must remain stable and serve as reference points for the production process. Typically the `main` branch. Contains the latest stable and production-ready code. Only approved, tested code is merged here. Releases are marked using tags, such as v1.0.0, typically using semantic versioning.
  Automated CI/CD pipelines trigger builds and deployments when code is merged.
- Ephemeral branches: avoid commits on persistent branches. The use of ephemeral branches isolates every single development in its branch avoiding any impact on other branches. The use of ephemeral branches allows to create merge requests, Enable code reviews before merging.

Maintain clean and structured code. Ensure that only validated changes reach persistent branches. Different types of ephemeral branches:

- feature: used to develop new functionalities before merging into main. Branch Naming Convention: `feature/ID-description` or `ID-feat-description`, where ID is the GitLab ID of the issue associated to the new feature. Workflow:
  1. Branch is created from dev
  2. Code is developed, including unit/integration tests
  3. Merge Request is opened for team review and approval
  4. Once approved, the branch is merged back into main and deleted(*)
- hotfix: intended to fix critical bugs found in production (main). Branch Naming Convention: `hotfix/ID-bugDescription` or `ID-fix-bugDescription` where ID is the GitLab ID of the issue associated to the bug. Workflow:
  1. Branch is created from main
  2. Bug is fixed and tested
  3. Merge Request is opened for team review and approval
  4. Once approved, the branch is merged back into main and deleted(*)
  5. A new tag and release is created on main branch

Best practices

- Keep main stable
- Only merge approved code
- Use meaningful Merge Requests
- Provide clear descriptions and references, add images if GUI is impacted
- Follow a strict naming convention for branches
- Delete ephemeral branches after merging

2.3.2. Code reviews

A Code Review is the process of examining someone else's code, usually before it is merged into the main branch. It is an essential step in collaborative development that helps ensure code quality, correctness, and consistency.

Code Reviews are often performed through Merge Requests, where team members can read the changes, leave comments, and suggest improvements. This process improves not only the code, but also helps developers learn from each other, share knowledge, and maintain a clean, understandable codebase.

Code review has several goals:

- Help catch bugs before code is merged
- Improve code readability and maintainability
- Ensure adherence to coding standards
- Promote knowledge sharing within the team
- Encourage communication and constructive feedback

A team should define clear coding standards and conventions defining naming, formatting, file structure. Review policies help decide what must be checked in every review: tests, documentation, error handling.

Checklists can be created to ensure consistency and completeness. For instance:

- Does it include tests?
- Are all public methods documented?

These rules help reviewers know what to focus on. They also help authors write code that meets expectations from the start

During the code review there are two roles:

- **Author**: the person who wrote the code and opened the Merge Request
- **Reviewer**: the person who reviews the code and provides feedback

Both roles are essential to ensure quality and shared responsibility. Communication between author and reviewer should be respectful and focused on the code. Code reviews are a form of professional communication, therefore respectful behavior helps avoid conflicts and misunderstandings. Even written feedback can be easily misinterpreted without proper tone so special care should be devote to adopting a proper one. Reviews should improve the code, not judge the person! A good environment encourages learning, trust and collaboration.

Etiquette makes the process more effective and enjoyable for everyone

- Etiquette for the Author:
  - Be open to feedback and don't take it personally
  - Clarify your choices respectfully if questioned
  - Fix issues quickly and explain your changes when replying
  - Resolve discussions only when feedback has been addressed
  - Thank reviewers for their time and input
- Etiquette for the Reviewer:
  - Focus on the code, not the person
  - Be respectful and constructive in your comments
  - Ask questions instead of making demands, e.g., "Why not use X here?" instead of "This is wrong"
  - Suggest improvements clearly and politely
  - Avoid minor comments that don't affect clarity or correctness
  - Approve only when the code meets the agreed standards

2.4. Gitlab Features

2.4.1. Issues

GitLab **Issues** are a simple way to track tasks, bugs, and new features. They help teams organize work, assign responsibilities, and document decisions.

Creating an Issue allows developers to describe what needs to be done before writing code. Issues can be linked to branches and Merge Requests, making collaboration clear and traceable.

GitLab allows to create a branch and a Merge Request directly from the Issue; the branch is named correctly and clearly and the MR is automatically opened from the new branch, user chooses the destination branch (typically `main`). The issue, branch, and MR are automatically connected ,and when the MR is merged, the issue is closed automatically

Creating a GitLab issue

- Go in the "Plan" section in the repository
- Choose the "Issues" subsection in the "Plan" tab
- Click "New Issue"
- Add a clear title and descriptive explanation

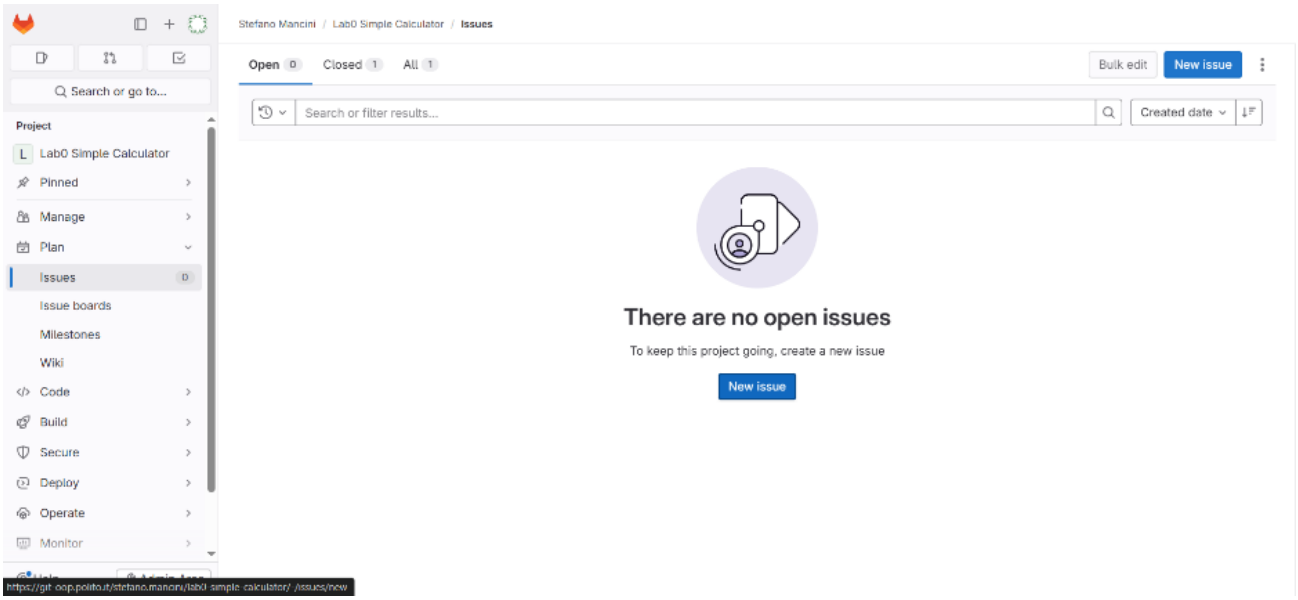


Figure 2.27.: Create GitLab issue

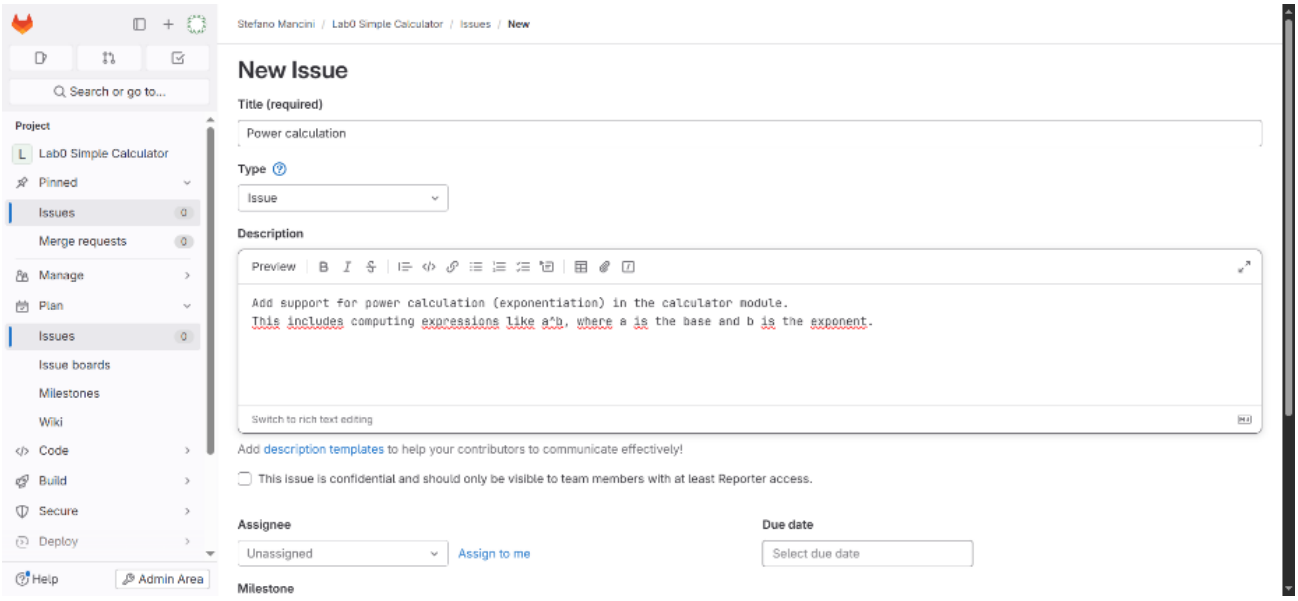


Figure 2.28.: Create GitLab issue

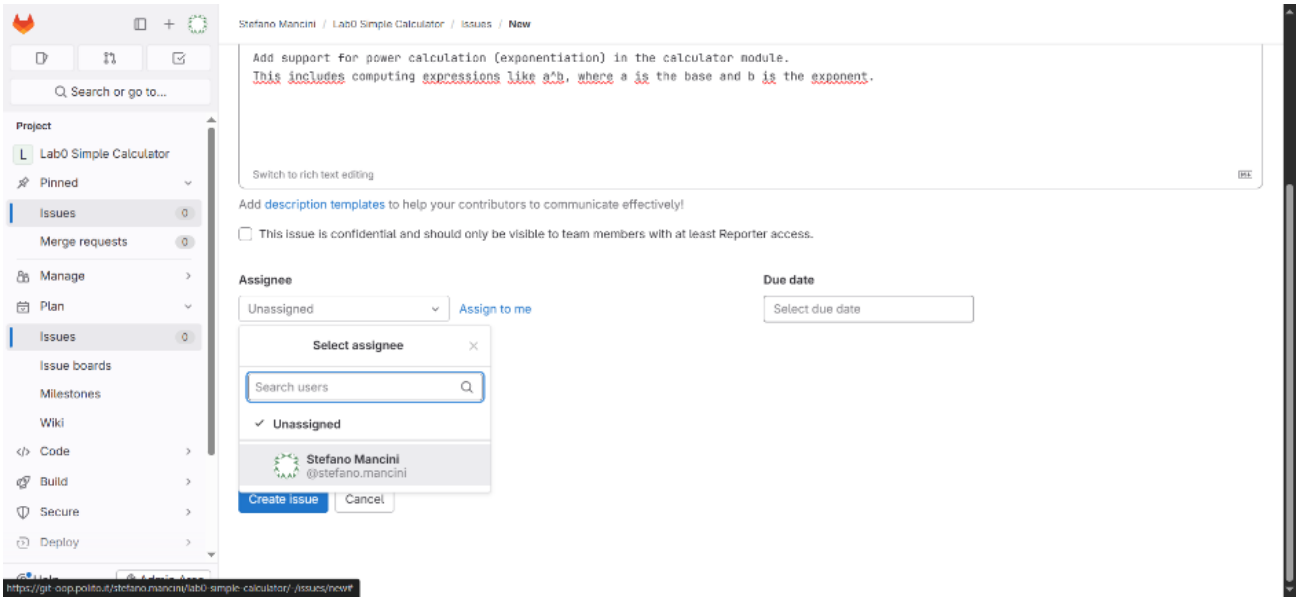


Figure 2.29.: Create GitLab issue

The GitLab issue can be furhter configured by providing the following (optional) details:

- Assignee: it can be assigned to yourself or a teammate,
- Labels: various labels can be added to categorize the task,
- Due date: a deadline for resolving the issue,
- Milestone: the issue can be linked to a milestone in the development plan.

2.4.2. Merge requests

A **Merge Request (MR)** is a GitLab feature (GitHub calls them Pull Request) that allows developers to propose changes made in one branch to be merged into another.

It is a collaborative tool that lets the team review the code before it becomes part of the main project, discuss changes and suggest improvements and track what has been modified and why.

Merge Request status allow understaning the stage of the work:

- draft is the initial state when work is still being performed,
- open is the state when work is done and reviewing can start,
- approved is the state after the code review and possible further changes, when the work related to the MR is considered complete,
- merged after the work, contained in a branch has been merged into the main branch.

When the MR is merged, it automatically closes the related issue.

A Merge Request is usually created created from an existing issue. Although it can also be created manually from an existing branch. If the description of the merge request mentions an issue it is linked to that issue and merge will close it. For instance, adding "Closes #7" in the description allows the automatic closing of the issue 7.

Creating a Merge Request from an issue takes the user directly to the Merge Request configuration step. This operation creates a branch too, and opens the Merge Request automatically from the newly created branch towards the default branch.

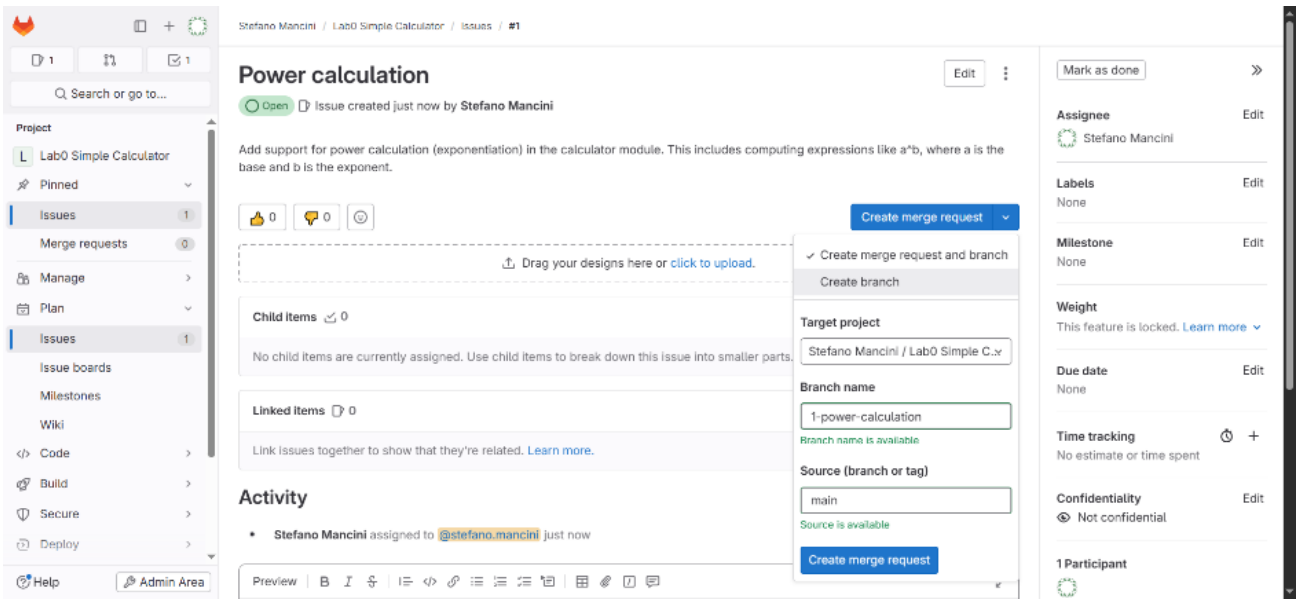


Figure 2.30.: Create GitLab issue

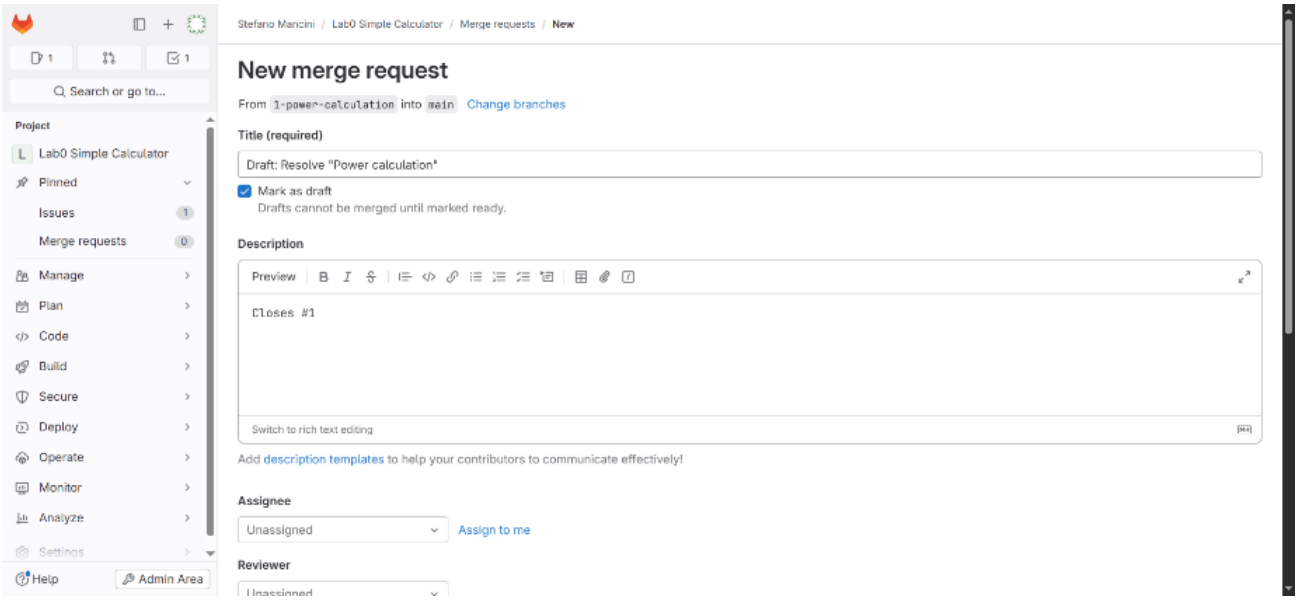


Figure 2.31.: Create GitLab issue

The Merge Request can be configured by providing a few details:

- An assignee can be selected to indicate who is responsible for the work
- One or more reviewers can be added to request feedback
- A milestone can be assigned to link the MR to a broader objective
- Labels can be used to categorize or prioritize the MR
- The MR can be set to automatically delete the source branch after merging
- A squash option can be enabled to combine all commits into a single one when merging
- The MR can be marked as draft to indicate it is not ready for review yet

2.4.3. Code reviews

GitLab offers two main ways to review code in a Merge Request:

- adding inline comments and discussions
- submitting a structured review with approval

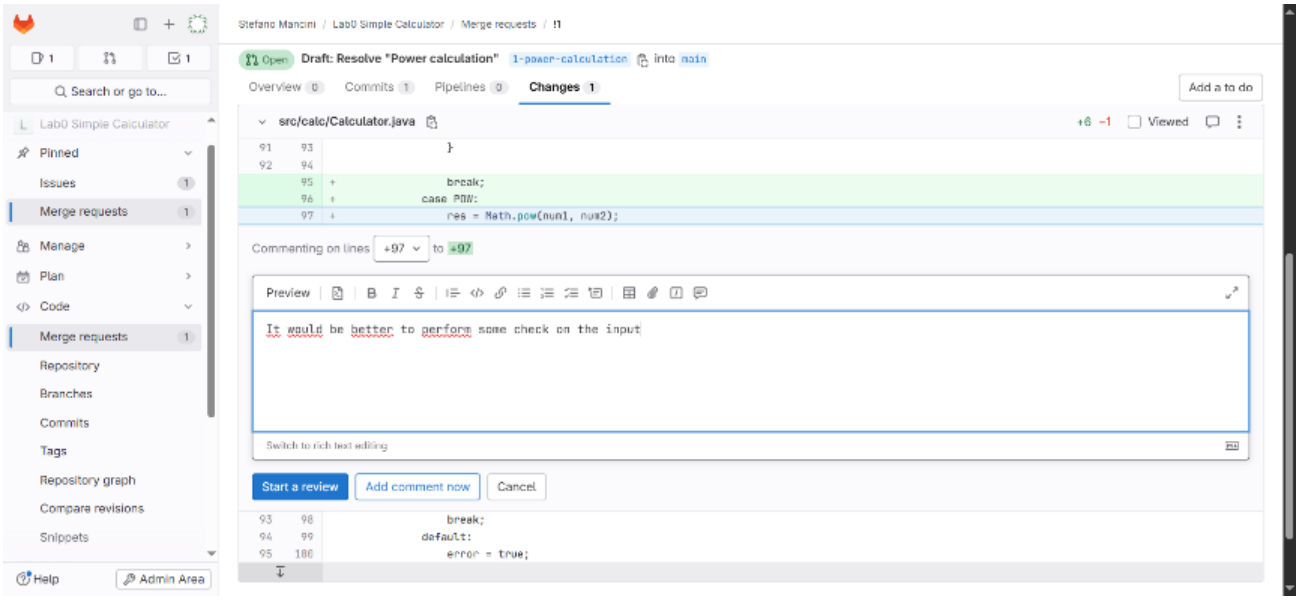


Figure 2.32.: Create GitLab issue

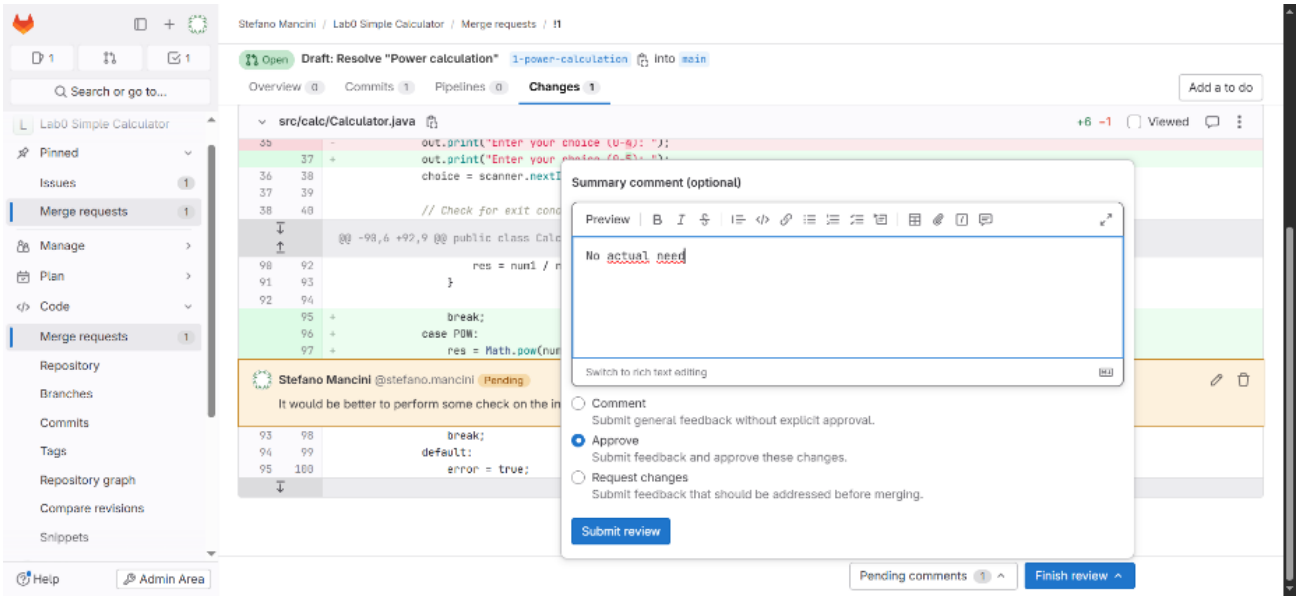


Figure 2.33.: Create GitLab issue

Inline comments are simple and flexible, and can be used at any time.

A formal review groups feedback and provides a clear approval status.

Submitting a review is optional: a Merge Request can still be merged even without a formal review. This feature is useful for teams that want a more structured review process.

The Merge Request interface shows the differences between the two branches. Code changes can be reviewed line by line. Inline comments can be added to specific lines of code. Discussion threads can be

started to ask for clarification or suggest improvements. This enables a structured and collaborative code review process.

Inline comments can be added directly on specific lines in the "Changes" tab. Comments are useful for asking questions, suggesting improvements, or clarifying choices. Each comment can be marked as a discussion, allowing replies and resolutions. Discussions stay open until explicitly resolved. Comments can be posted immediately, without starting a full review.

GitLab allows reviewers to submit a formal review after examining the code. The review can include multiple comments grouped together.

At the end of the review, the reviewer can choose to:

- Approve the merge request
- Request changes with feedback
- Leave comments without approval

A full review provides a clear review status for the author and the team

2.4.4. MR merge

When an MR is finally approved and gets merged into the destination branch. The changes from the source branch are integrated into the target branch.

If enabled, the source branch is automatically deleted to keep the repository clean[2] If selected, all commits are squashed into a single commit before merging[3]

The Merge Request status is updated to "Merged"- If the MR references an Issue (e.g., "`Closes #12`"), the Issue is automatically closed. The code in the target branch now includes the approved changes and is ready for further development or release.

[2] in the team projects of the course, commits must be preserved as they are, so not squashed and the branch not deleted. This is the default configuration for such projects.

[3] in the team projects of the course, commits must be preserved as they are, so not squashed and the branch not deleted. This is the default configuration for such projects.

3. Build Management and Continuous Integration

The software must be eventually released, i.e. distributed outside the development activity, both internally – e.g. for testing – and to customers.

Build management concerns how software is constructed, using the appropriate configuration data, into a software package for delivery to a customer or other recipient such as a team performing testing ("Guide to the Software Engineering Body of Knowledge" 2024).

The build management is usually automated though tools such as make, ant, maven, gradle. In the context of Java applications the two most used build automation tools are Maven and Gradle.

In continuous integration (CI), software building is performed automatically when changes are committed to the repository. Tools monitor the project's repository and initiate a pipeline of steps to be undertaken every time a change iscommitted to a particular branch. This behavior may be combined with steps to validate coding standards via automated static analysis, execute unit tests and determine code coverage metrics, or generate documentation from the source code.

3.1. Maven

Maven is a build automation and project management tool designed to standardize how Java projects are built, tested, and deployed. Maven automates repetitive tasks such as compiling code, downloading dependencies, packaging applications, and running tests—all with a few simple commands. It makes the build process easy by providing a uniform build system and encouraging better development practices.

Maven automates the key phases of the build process:

- Compiling Java source code into bytecode.
- Running unit tests to verify correctness.
- Packaging the compiled code into JAR or WAR files.
- Deploying the resulting artifacts to local or remote repositories.

For instance with the following command it is possible to clean possible old files, compile the coce, run the tests and install the package in a local repository, available to ather project that would need it:

```
mvn clean install
```

Maven is based on the following basic principles:

- Convention over configuration: Maven assumes a standard project structure and common naming conventions, so you don't need to specify every detail. The default for a project also includes the lifecycles and the relative plugins.

- Declarative execution Instead of writing scripts that describe how to build your project, Maven uses a declarative approach: the `pom.xml` configuration file describes what the project is and what it needs. Such information is used to perform the correct build steps.
- Dependency Management Maven automatically downloads required libraries (and their dependencies) from online repositories. Dependencies are declared in the `pom.xml` and Maven ensures they are available during compilation, testing, and runtime.
- Lifecycle Management Every Maven build follows a defined lifecycle, a sequence of standard phases like compile, test, package, and install. Running one command (e.g., mvn package) triggers all necessary earlier phases in order.
- Plugin Architecture Almost all Maven functionality is provided through plugins. Each plugin performs a specific task.
- Reproducibility Because everything – dependencies, versions, and build instructions – is explicitly defined in `pom.xml`, anyone can reproduce the same build on any machine. This makes projects portable, reliable, and easy to integrate into automated build systems.

3.1.1. Project structure

Maven works assuming a standard directory layout for projects:

- root folder contains the `pom.xml` configuration file and other general files,
- the `src` folder contains the source files and the resources,
- the `target` folder will contain the results fo the build process.

The root foder usually includes a `README.md` that, by default, is rendered by the repository web portals on the initial page.

A more complex project uses the following structure:

Table 3.1.: Standard Directory Layout

Folder	Description
`src/main/java`	Application/Library sources
`src/main/resources`	Application/Library resources
`src/main/filters`	Resource filter files
`src/main/webapp`	Web application sources
`src/test/java`	Test sources
`src/test/resources`	Test resources
`src/test/filters`	Test resource filter files
`src/it`	Integration Tests (primarily for plugins)
`src/assembly`	Assembly descriptors
`src/site`	Site
`pom.xml`	The project object model with configuration
`LICENSE.md`	Project's license
`NOTICE.md`	Notices and attributions required by libraries that the project depends on
`README.md`	Project's readme

Maven also adopts a standard naming conventions where the name of the component is followed by a `-` (dash), the version number, and the extension. E.g., `commons-logging-1.2.jar` vs. `commons-logging.jar`

Please note that the course adopts a simplified structure for the base projects.

The main benefits of a standard project layout are:

- consistency: every project looks the same, making it easy to navigate unfamiliar codebases;
- automation: Maven automatically knows where to find and put files thus leveraging convention over configuration;
- portability: continuous integration tools (e.g., Jenkins, GitHub Actions, GitLab Pipelines) can build any Maven project without additional configuration.

3.1.2. Lifecycle

Maven organizes the build process into a series of **lifecycles**, which are structured sequences of **phases** that define what happens and in what order during a build. A **lifecycle** is an ordered sequence of phases, Maven defines three standard lifecycles:

- *default* lifecycle `mvn build`
- *clean* lifecycle `mvn clean`
- *site* lifecycle `mvn site`

Each lifecycle can be run independently (e.g., `mvn clean`, `mvn site`) or combined (e.g., `mvn clean install`).

The default *build* lifecycle is made up of the following main phases:

- `validate`: validate the project is correct and all necessary information is available
- `compile`: compile the source code of the project
- `test`: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package`: take the compiled code and package it in its distributable format, such as a JAR.
- `verify`: run any checks to verify the package is valid and meets quality criteria
- `install`: install the package into the local repository, for use as a dependency in other projects locally
- `deploy`: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Figure 3.2.: Maven build lifecycle default phases

The phases of the other lifecycles are:

The Clean lifecycle is used to clean the workspace by deleting generated files (e.g., compiled classes, JARs) in the `target/` directory:

- Pre-clean: preparation before clearning,
- Clean: deletes output files,
- Post-clean: possible tasks performed after cleaning.

The Site lifecycle generates project documentation, reports, and a website using project metadata:

- Pre-site: prepare documentation resources;
- Site: generates documentation (HTML reports, dependency graphs, etc.);
- Post-site: perform follow-up tasks,
- Site-deploy: publish the generated site to a server.

Maven can be invoked passing the name of a lifecycle (e.g. `mvn build`), in that case all the phases of the lifecycle are executed in orded.

It is also possible to invoke maven with the name of a phase (e.g., `mvn test`), in this case it will first execute all preceding phases in the lifecycle in order, ending with the phase specifiedon the command line.

An example usage of Maven invocation:

```
mvn clean install
```

This is the most common Maven command for building and installing a project locally:

- uses the `clean` lifecycle to remove old build files.
- executes the `default` lifecycle phases from the first up to the `install` phase

There can be *goals* can be attached to a lifecycle phase. As Maven moves through the phases in a life cycle, it will execute the goals attached to each particular phase

3.1.3. POM

Everything in Maven is defined in a declarative fashion using Maven's Project Object Model (POM) that is defined in the `pom.xml` file.

The `pom.xml` file contains the configuration and the metadata for the project. It replaces the need for complex build scripts by providing a declarative description of the project's structure and requirements. Maven uses the information in this file to:

- identify the project uniquely (through group and artifact identifiers).
- locate and manage dependencies.
- bind plugins to lifecycle phases.
- define build options, reporting tools, and repositories.

The basic structure of `pom.xml` is as follows

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/mav
    <modelVersion>4.0.0</modelVersion>
    <groupId>it.polito.oop.samples</groupId>
    <artifactId>samples</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>OOP Samples</name>
    <description>Sample code for the OOP Course</description>

    <properties>
        <!-- Property definitions go here -->
    </properties>

    <build>
        <plugins>
        <!-- Plugin definitions go here -->
        </plugins>
    </build>

    <dependencies>
        <!-- Dependency definitions go here -->
    </dependencies>

</project>
```

The mandatory elements, in addition to the initial `<project>` tag with the relative namespace declarations, are:

- `modelVersion`: Specifies the POM model version and is always `4.0.0` for modern Maven files,
- `groupID`: defines the id of the group, it is typically linked to a company or a unit of the company, it often corresponds to the Java package (or prefix of the packages) of the project, conventionally it follows the reverse internet name convention;
- `artifactID`: the name of the project;
- `version`: the current version number of the project.

Other optional – i.e., with a conventional default – parameters are:

- `packaging`: defines the output type (`jar`, `war`, `pom`, etc.).
- `name` and `description`: contain human-readable metadata.

The file may contain other additional sections:

- `<properties<`: defines reusable variables, such as Java version or encoding.
- `<dependencies>`: lists external libraries required for the build.
- `<build>`: contains configuration for plugins, build directories, and output settings.
- `<repositories>`: declares additional locations to fetch dependencies from.

The `<dependendencies>` block lists the libraries that are required by the project either for compilation or any other phase. In the above example the library `junit-jupiter` provided by `org.junit.jupiter` is mentioned, which is used to compile and run JUnit tests.

3.1.4. Dependencies

Dependencies define the external libraries your project needs to compile, test, or run.

Instead of manually downloading `.jar` files and managing paths, Maven automatically retrieves them from repositories based on the details provided in the `<dependencies>` section..

Example structure of dependency declaration:

```
<dependencies>
  <dependency>
    <groupId>group</groupId>
    <artifactId>artifact</artifactId>
    <version>version</version>
  </dependency>
</dependencies>
```

A dependency is described by three key identifiers:

- `groupId`: the organization or project the library belongs to,
- `artifactId`: the name of the library,
- `version`: the specific version to use,
- `scope`: determines when the specific dependency is made available in the classpath.

For instance, to include JUnit 5 as your testing framework, you can add the following dependency:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId> # <1>
    <artifactId>junit-jupiter</artifactId> # <1>
    <version>5.10.0</version> # <1>
    <scope>test</scope> # <2>
  </dependency>
</dependencies>
```

① the version `5.10.0` of the `org.junit.jupiter:junit-jupiter` artifact includes the full JUnit 5 API and engine.
② the `test` scope means JUnit will be available only during the testing phase, not in production builds.

Maven defines six distinct scopes for the dependencies:

- **compile** (default) these dependencies are available in all classpaths of a project.

- **provided** indicates you expect the JDK or a container to provide the dependency at runtime[1]. Such dependencies are added to the classpath used for compilation and test, but not the runtime classpath.
- **runtime** indicates that the dependency is not required for compilation, but is for execution. Maven includes a dependency with this scope in the runtime and test classpaths, but not the compile classpath.
- **test** indicates that the dependency is not required for normal use of the application, and is only required for the test compilation and execution phases. Typically this scope is used for test libraries such as JUnit and Mockito. It is also used for non-test libraries such as Apache Commons IO if those libraries are used in unit tests but not in the model code.
- **system** indicates that the dependency is required for compilation and execution. Maven will look for a jar at a specified path in the local file system and not download it.
- **import** indicates the dependency is to be replaced with the effective list of dependencies in the specified POM's `<dependencyManagement>` section.

All dependencies are declared inside the `<dependencies>` section of `pom.xml`. Maven reads this section, checks your local repository , and if the library isn't found locally, downloads it from the Maven Central Repository or another defined source.

Maven's **dependency management** system ensures all required versions (and their transitive dependencies) are resolved automatically. If multiple libraries depend on different versions of the same artifact, Maven applies a consistent conflict resolution strategy to select the correct one.

3.1.5. Plugins

Maven itself provides only the framework; the execution logic is is encapsulated into coherent modules called plugins. Maven coordinates the execution of plugins according to the phases of the lifecycles.

Each plugin provides one or more *goals*, which are individual tasks it can perform. When you run a Maven command, such as `mvn compile`, Maven executes the goals attached to the `compile` phase by triggering the execution of the plugins providing such goals.

Plugins can be executed automatically (because they are tied to a lifecycle phase) or manually by specifying them directly. For example `mvn compiler:compile` runs the `compile` goal from the compiler plugin explicitly.

The phases are generally fixed for the default lifecycles, each phase has a few predefined goals. Additional goals can be defined and added by plugins that are included in the configuration.

For instance the `build` lifecycle phases have the following goals:

Phase	Default Goal(s)	Common (default) plugin
`validate`	—	
`compile`	`compiler:compile`	`maven-compiler-plugin`
`test`	`surefire:test`	`maven-surefire-plugin`
`package`	`jar:jar` (for JAR projects)	`maven-jar-plugin`
	`war:war` (for WAR projects), etc.	`maven-war-plugin`

[1]For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope `provided` because the web container provides those classes.

Phase	Default Goal(s)	Common (default) plugin
`verify`	—	
`install`	`install:install`	`maven-install-plugin`
`deploy`	`deploy:deploy`	`maven-deploy-plugin`

Plugins are defined and configured inside the `<build>` section of the `pom.xml` file can add new plugins or ovverride default plugins.

Example structure of a plugin declaration

```
<build>
  <plugins>
    <plugin>
      <groupId>group</groupId>
      <artifactId>artifact</artifactId>
      <version>version</version>
      <configuration>
        <!-- Values for plugin parameter go here -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

For example, the default Java language version in the compiler plugin is 1.8. Compiling source code with Java version >= 9 requires a specific plugin (i.e., compiler > 3.6.0) and the configuration parameters indicating the version to be used by the compiler.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <source>25</source>
        <target>25</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This configuration ensures that the project is compiled using Java 25 version for the source language and generate bytecode compatible with version 25.

3.1.6. Properties

In the POM it is possible to define variables in a `<properties>` block, .e.g,

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>25</maven.compiler.source>
    <maven.compiler.target>25</maven.compiler.target>
</properties>
```

Later in the POM it is possible to recall the value of the property with the `${}` notation.

```
...
<target>${maven.compiler.target}</target>
...
```

3.1.7. Repositories

Java uses a standard way of deploying compiled code through `.jar` files. Therefore any time a plugin or an external library is required, Maven needs to find out where here does the dependency come from, and – specifically – where is the jar file?

The artifacts that constitute dependencies are stored in repositories. Repositories contain both libraries and plugin jars.

Usually the Maven dependency and plugin resolution procedure works using two repositories:

- a local repository and
- a remote repository.

Maven usually interacts with your local repository, but when a declared dependency (or plugin) is not present there, it searches all the remote repositories it has access to in an attempt to find what's missing.

Any dependency not present locally and found in a remote repository is downloaded to the local repository for later quick access (caching). Maven also downloads the dependency of the required artifacts, for instance if a library or plugin lists among its dependencies other libraries, then Maven proceeds to download them too, recursively.

By default the repositories are:

- Local : `/$HOME/.m2`
- Remote: `http://mirrors.ibiblio.org/pub/mirrors/maven2/` or `https://repo.maven.apache.org/mave`

The local repository dependencies and plugins are hosted in a folder called `repository` (e.g., `/$HOME/.m2/repository`). It contains one folder per group-id component. So for instance if the group-id is `org.apache.maven` there will be the three nested folders: `org` containing `apache` containing `maven`. In that latter folder there is one sub-folder for each artifact-id and, inside that, one folder for each specific version.

For instance a typical dependency that is present in virtually every Java project is on the JUnit framework (which is not part of the JDK library)

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId> # <1>
        <artifactId>junit-jupiter</artifactId> # <2>
        <version>5.10.0</version> # <3>
        <scope>test</scope> # <4>
    </dependency>
</dependencies>
```

① the group id of the library
② the artifact id
③ the required version of the artifact
④ the scope where the dependency will be made available by Maven

The local repository (`/$HOME/.m2/repository`) will then contain

```
org # <1>
+- junit # <1>
   +- jupiter # <1>
      +- junit-jupiter # <2>
          +- 5.13.4 # <3>
            +- junit-jupiter-5.13.4.jar # <4>
            +- junit-jupiter-5.13.4.jar.sha1 # <5>
            +- junit-jupiter-5.13.4.pom # <6>
            +- junit-jupiter-5.13.4.pom.sha1 # <7>
```

① group id folder hierarchy
② artifact id folder
③ the version folder
④ the jar containing the library code
⑤ the signature of the jar file
⑥ the pom of the library (containing e.g. cascade dependencies)
⑦ the signature of the pom

3.1.8. Project creation

Maven creates new projects using archetypes, which are predefined (plugin) templates capable of generating a standard directory structure and a working `pom.xml`. This makes it easy to start a new project that follows Maven's conventions.

The following command line can create a new project:

```
mvn archetype:generate \ # <1>
  -DgroupId=com.example \ # <2>
  -DartifactId=my-app \ # <3>
  -DarchetypeArtifactId=maven-archetype-quickstart \ # <4>
  -DinteractiveMode=false                                    ⑤
```

① calls the creation goal of the default archetype plugin;
② `groupId` identifies your organization or project (e.g., `it.polito.oop`);
③ `artifactId` is the name of the project or module (e.g., `my-app`);
④ `archetypeArtifactId` pecifies which template to use; `maven-archetype-quickstart` creates a simple standard Java project;
⑤ `interactiveMode=false` prevents Maven from prompting for additional input, allowing the command to run automatically.

3.1.8.1. Simplified base project

The OOP course at Politecnico uses a simplified version of the project layout that is simpler to understand and navigate w.r.t. the default project structure.

The archetype used to genereate the course project is `it.polito.oop.base-project`. Its simplified structure is:

```
project
+-  pom.xml
+-  README.md
+-  .gitignore
+-  src ...
    +-  App.java
+-  test ...
    +-  test
        +-- TestApp.java
```

It is possible to generate a new project (in interactive mode) with

```
mvn archetype:generate \
-DarchetypeGroupId="it.polito.oop" \
-DarchetypeArtifactId="base-project"
```

Alternatively, without interactive mode:

```
mvn archetype:generate \
-DarchetypeGroupId=it.polito.oop \
-DarchetypeArtifactId=base-project \
-DgroupId=it.polito.samples \
-DartifactId=my-app \
-DinteractiveMode=false
```

After the project has been created (either with the simplified course structure or the default one), it is possible to write the code and test it. And eventually pacakge the resulting code into a `.jar` file:

```
mvn package
```

In this case `package` is a phase, thus Maven executes every phase in the sequence up to and including the one provided. The results of the compilation as well as of the eventual pacakging are placed in the `target` folder.

Is is possible to execute the newly compiled and packaged JAR with the following command:

```
java -cp target/my-app-1.0.0.jar it.polito.samples.App
```

3.2. Continuous Integration

> Continuous Integration is a software development practice where each member of a team merges their changes into a codebase together with their colleagues changes at least daily. Each of these integrations is verified by an automated build (including test) to detect integration errors as quickly as possible. (Fowler 2024)

CI tasks are executed in what GitLab calls a *Pipeline*, and GitHub *Workflows* or *Actions*. In this book we will examine the GitLab version, although the concepts are very similar and can be translated from one context to the other pretty straightforwardly.

3.2.1. Pipelines and jobs

Gitlab Pipelines can run automatically for specific events, like when pushing to a branch, creating a merge request, or on a schedule. When needed, they can also be run manually.

Pipelines are composed of:

- **Jobs** that execute commands to accomplish a task. For example, a job could compile, test, or deploy code. Jobs run independently from each other, and are executed by runners.
- **Stages**, which define how to group jobs together. Stages run in sequence, while the jobs in a stage run in parallel. For example, an early stage could have jobs that lint and compile code, while later stages could have jobs that test and deploy code. If all jobs in a stage succeed, the pipeline moves on to the next stage. If any job in a stage fails, the next stage is not (usually) executed and the pipeline ends early.

If stages are not explicitly defined, the default pipeline stages are:

- `.pre`
- `build`
- `test`
- `deploy`
- `.post`

GitLab pipelines record a log of the execution of the pipeline. In the case of a Java project build using maven, it collects the output of the maven command. The result is similar to what is observe in the terminal locally when running the same command.

In addition Gitlab shows the content of reports or other artifacts that are produced by the jobs in the pipeline.

The `junit` report collects JUnit report format XML files. The collected Unit test reports upload to GitLab as an artifact.

GitLab can display the results of one or more reports in:

- The merge request Test summary panel.
- The pipeline Tests tab.

The details about the jobs are defined in the `.gitlab-ci.yml` file.

```
variables:
  MAVEN_CLI_OPTS: >-
    --batch-mode
    --fail-at-end
    --show-version

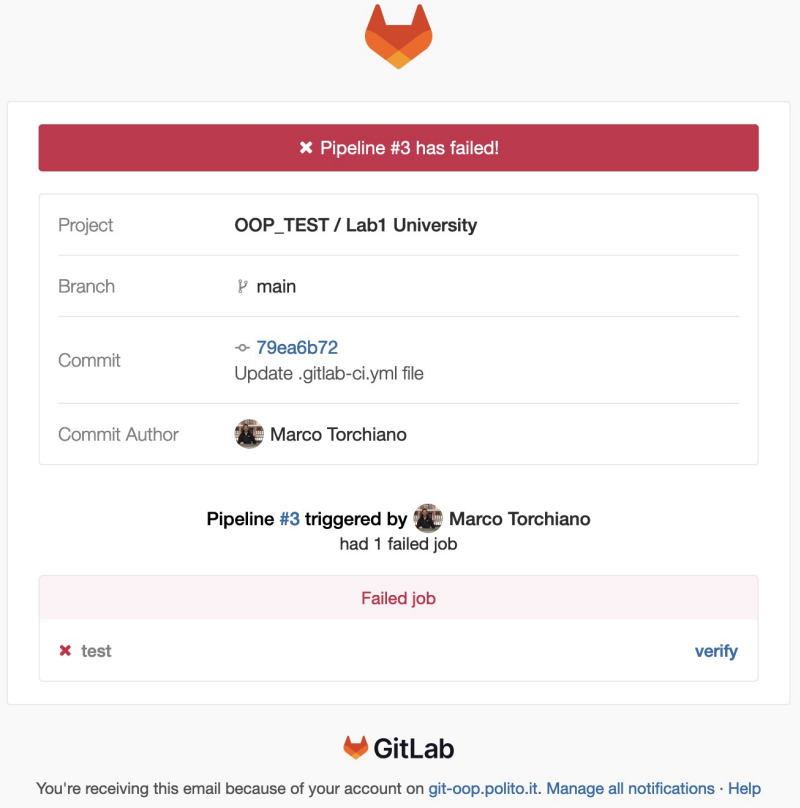
verify:
  stage: test
  script:
    - 'mvn $MAVEN_CLI_OPTS test'
  artifacts:
    when: always
    reports:
      junit:
        - target/surefire-reports/TEST-*.xml
```

3.2.2. Test results

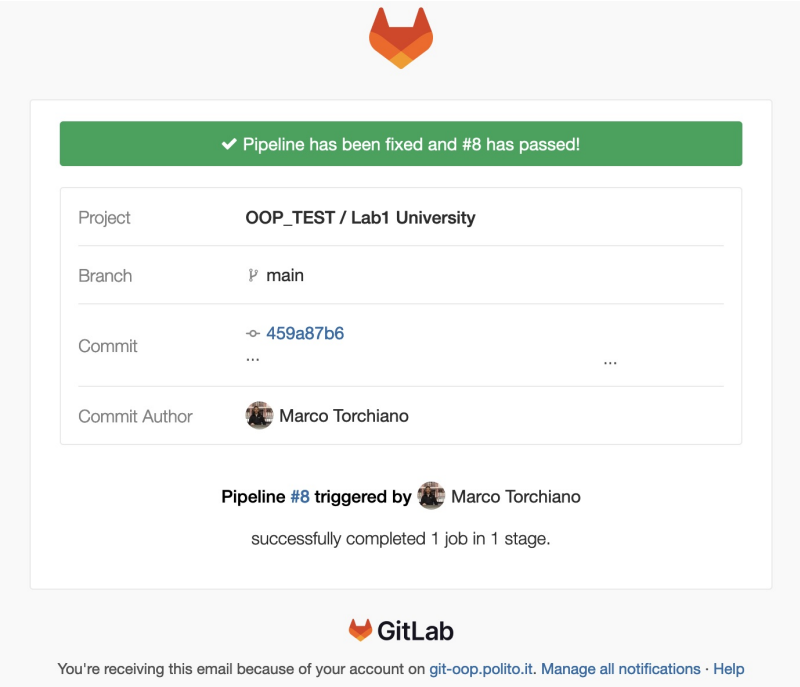
3.2.2.1. Execution notification

When a pipeline is executed, the GitLab server sends a notification email with the outcome to the repository owner.

- It can be a failure, for example:



- Or a success, for example:



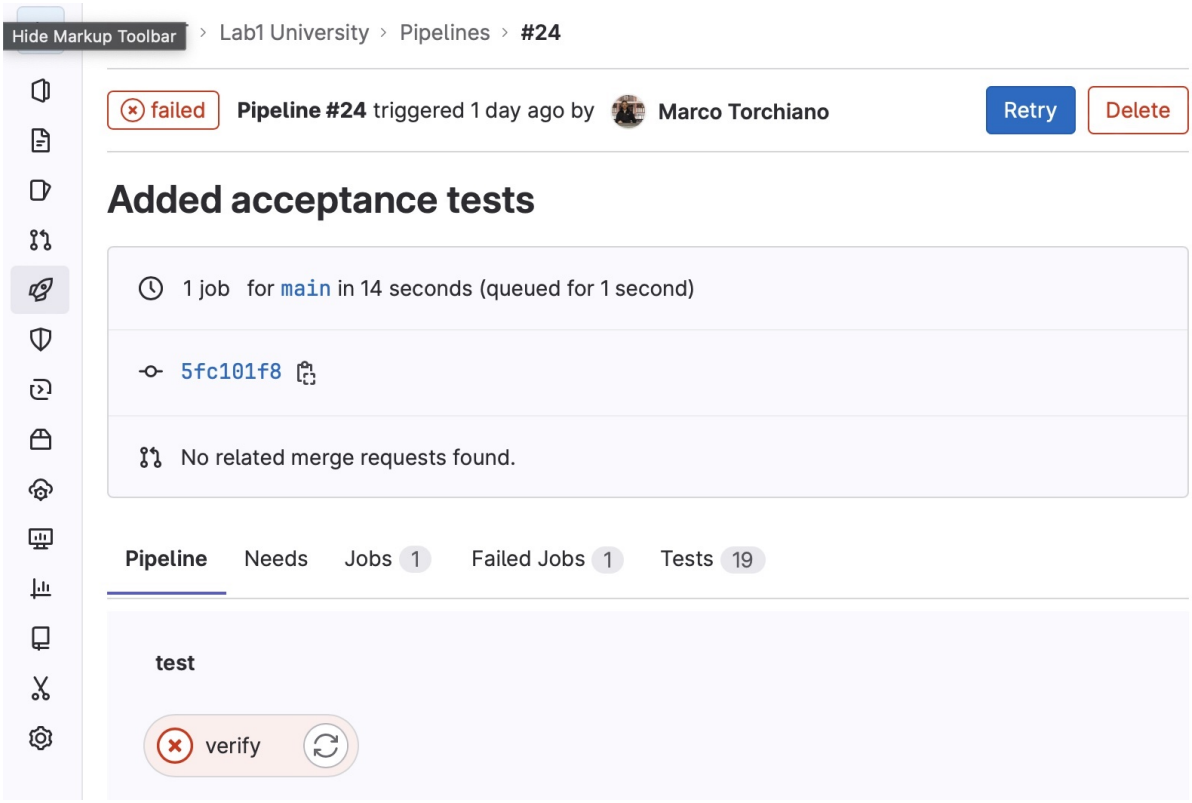
If you click on the pipeline number, e.g. `#8`, you will access the pipeline visualization online at the GitLab site.

3.2.2.2. Pipeline visualization

You can view details about the outcome of a pipeline in two ways:

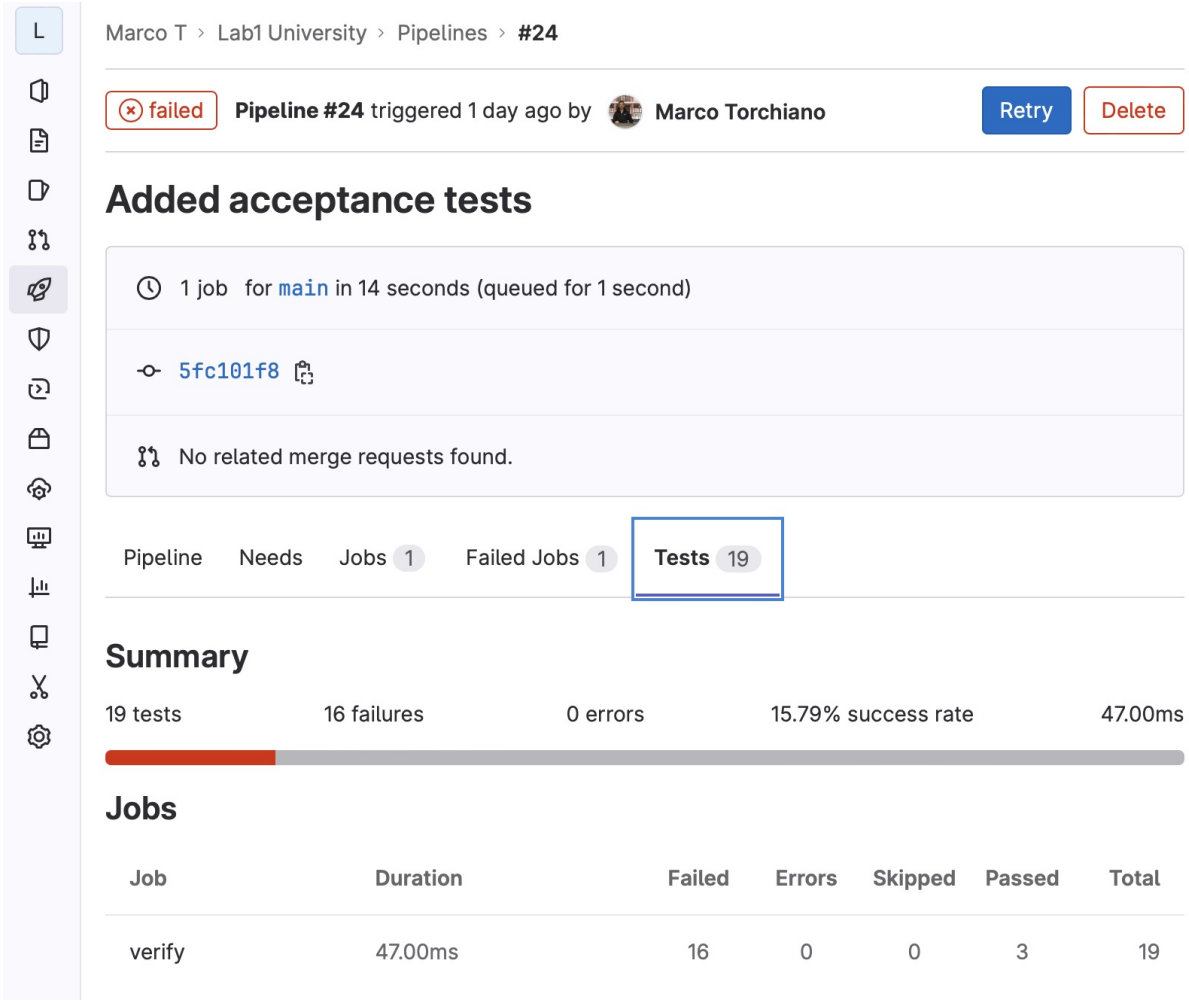
- by clicking on the link in the Execution notification email,
- from the project page:
  - in the menu on the left margin *CI/CD* and then *Pipelines* you see the list of executed pipelines
  - by clicking on the `passed` or `failed` status label.

You get to the pipeline view page:

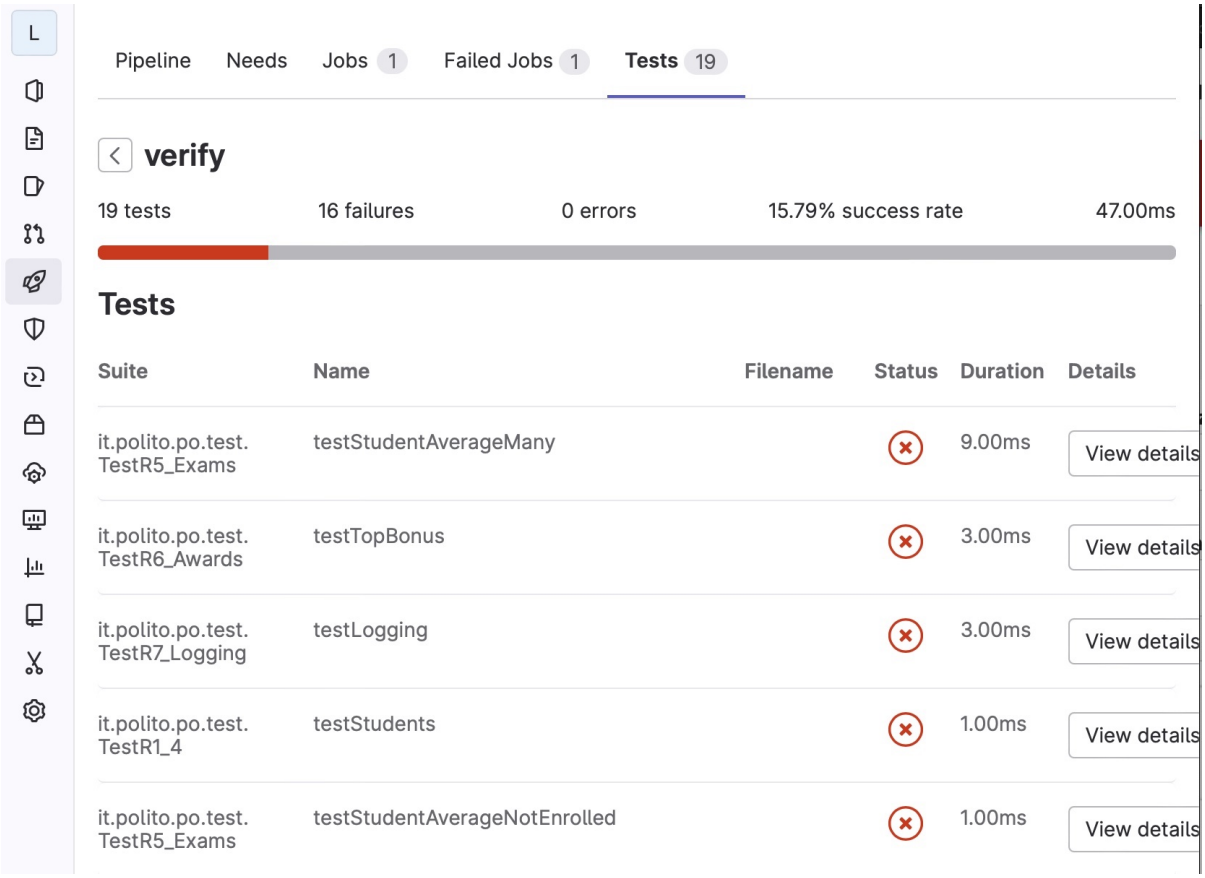


Click on the `verify` job to see the log of the build and test execution of the project via Maven: useful to understand any errors but difficult to interpret.

You may want to click on the `Tests` tab to view the summary of tests run:



Clicking on the test row shows details with a list of individual test methods and their respective outcome.



3.2.3. Understand failures

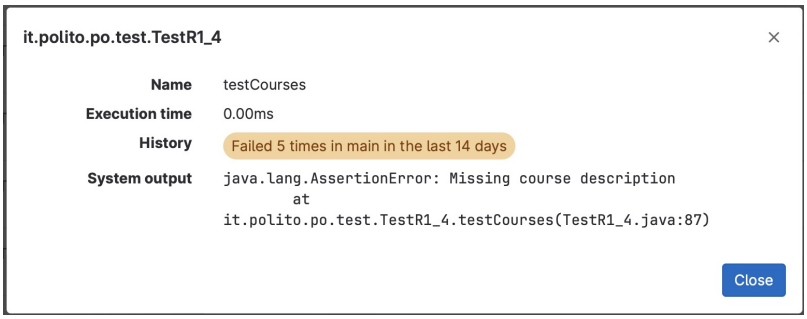
A the macro level it is important to understand wether the pipeline passed or failed.

A failing pipeline typically blocks a merge request, e.g. when using the GitFlow approach. Viceversa a passing pipeline gives green light to the merge.

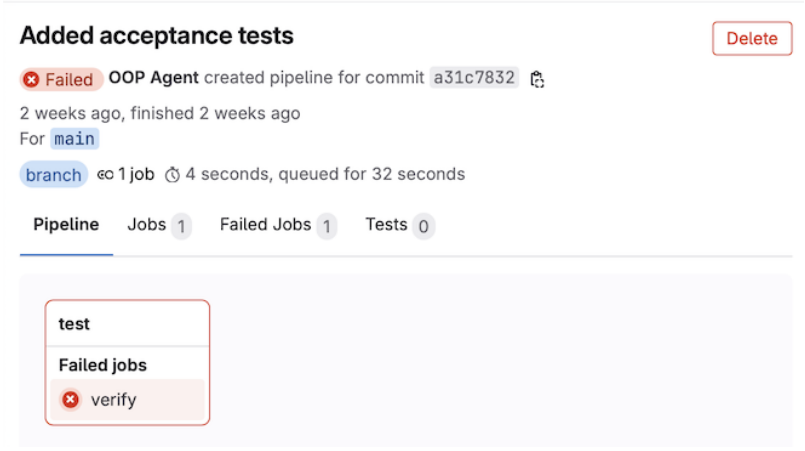
There are two main possible resons for a pipeline failure:

- some tests failing,
- compilation failure.

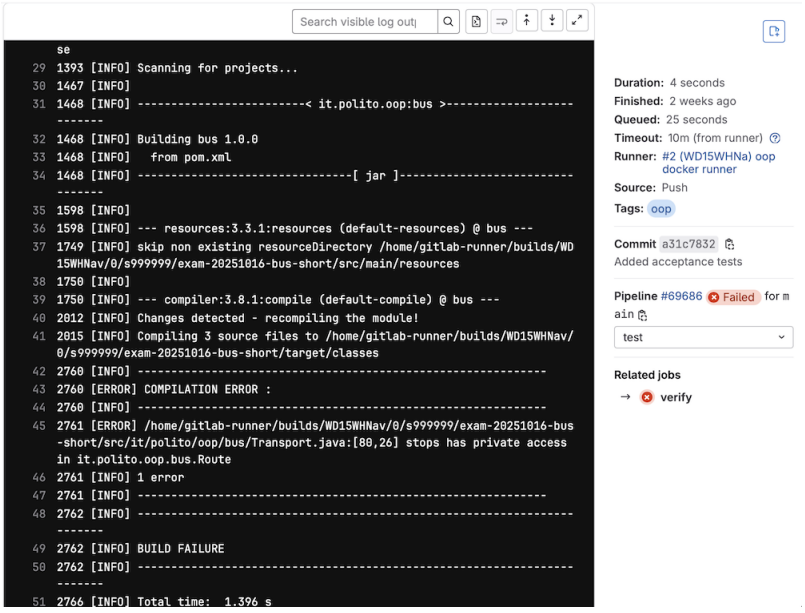
The typical case is when some tests fail (either due to failures or errors detected by JUnit). Clicking on the `Details` button shows the detail of the failed (or passed) test:



Another case occurs when a compilation error occurs. In this case no test can be executed an therefore tests is 0:



In this case to understand better what is the cause of the failure it is possible to inspect the detailed log of the pipeline. By clicking on the failed job, a window appears with the job details and the full text log of the pipeline execution.



In the sample log shown in the image above, a compilation error is reported.

Part II.

Java Language

4. Java Environment

Learning objectives

- Understand the basic features of Java
- What are portability and robustness?
- Understand the concepts of bytecode and interpreter
- What is the JVM?
- Learn few coding conventions

4.1. Java Timeline

The Java language originated in 1991 and then continued evolving up to the present day.

- 1991: SunMicrosystems develops a programming language for cable TV set-top boxes
  - Simple, OO, platform independent
- 1994: Java-based web browser (HotJava)
  - The concept of "applet" appears
- 1996: First version of Java (1.0)
- 1996: Netscape supports Java
  - Java 1.02 released,
- 1997: Java 1.1 released, major leap over for the language
- 1998: Java 2 platform (v. 1.2) released (libraries)
- 2000: J2SE 1.3
  - platform enhancements
  - HotSpot JVM
- 2002: J2SE 1.4 (several new APIs), e.g.
  - XML
  - Logging
- 2005: J2SE 5.0 (Language enhancements)
  - Generics
- 2006: Java SE 6 (Faster Graphics),
  - goes open source
- 2010: Acquisition by Oracle

- 2011: Java SE 7 (I/O improvements)
- 2014: Java SE 8 (Lang evolution, LTS)
  - Lambda expressions
  - Functional paradigm
- 2017: Java 9 released
  - Modularization (Jigsaw),
  - jshell (REPL)
- 2018: Java 10, Java 11 (LTS)
  - Local var type inference
- 2019: Java 12, Java 13
  - Switch expressions
  - Text blocks
- 2020: Java 14, Java 15
  - Text blocks
- 2021: Java 16, Java 17 (LTS)
  - Records
  - Jpackage
- 2022/23: Java 18, Java 19, Java 20
  - Simple web server
  - Virtual threads
- Latest release: Java 25(LTS) September 2025
  - Compact Source Files and Instance Main Methods

For more details see: https://www.java.com/releases/

An exhaustive list of the changes introduced in each version is available here: https://docs.oracle.com/en/java/javase/25/language/java-language-changes-release.html.

Since Java 10 a new regular release process has been put in place with a well defined versioning scheme. A new version is released every six months (in March and September), update releases every quarter, and a long-term support (LTS) release every three years. LTS relesases are usually supported for eight years (e.g. JDK 25 will be supported until September 2033).

4.2. Java features

OO languages provide the constructs to support the development of code based on the Object-Oriented approach.

The Java constructs allow to define define classes (that constitute types) in a hierarchical way (using inheritance), create/destroy objects dynamically, send them messages (w/ dynamic binding). Java does

not provide procedural constructs (it's close to a pure OO language): there are no functions, class methods only, no global vars, class attributes only.

A summary of the key Java features follows:

- Platform independence (portability)
- Write once, run everywhere
- Translated to intermediate language (bytecode)
- Interpreted (with optimizations, i.e. JIT)
- High dynamicity
- Run time loading and linking
- Robust language, less error prone
  - Strong type model and no explicit pointers
- Compile-time checks
- Run-time checks
  - No array overflow
- Garbage collection
  - No memory leaks
- Exceptions as a pervasive mechanism to check errors
- Shares many syntax elements w/ C++
  - Learning curve is less steep for C/C++ programmers
- Quasi-pure OO language
  - Only classes and objects (no functions, pointers, and so on)
  - Basic types deviates from pure OO…
- Easy to use
- Supports "programming in the large"
  - JavaDoc
  - Class libraries (Packages)
- Lots of standard utilities included
  - Concurrency (thread)
  - Graphics (GUI) (library)
  - Network programming (library)
  - socket, RMI
  - applet (client side programming)

4.2.1. Classes

Java has one first level concept: the class

The minimal syntactically correct code is:

```
public class First {
}
```

The source code of a class sits in a `.java` file having the same name, thus the above `First` class definition shall be placed in a file named `First.java`. The general rule is one file per class; it is enforced automatically by most IDEs. It is important to note that class-filename is a case-wise correspondence,they must match exactly.

4.2.2. Methods

In Java there are no functions, but only methods within classes. The syntax of methods is very similar to that of C functions although they are defined within the scope of a class and work in the context of objects.

The execution of a Java program starts from a special method:

```
public static void main(String[] args){ ... }
```

Please note the differences w.r.t. C

- return type is `void`,
- there is not `argc` argument,
- the `args` argument correspond the the C `argv` but, notably, `args[0]` is the first parameter on the command line (not the program name).

Since Java 25 it is possible to omit the `public static` part, provided the class has a public constructor without pararameters, thus having a *launchable main method*.

4.3. Java Ecosystem

The Java Ecosystem is made up of

- Java language
- Java platform
  - Java Virtual Machine (JVM)
  - Class libraries (API)
  - Software Development Kit (SDK)

The Java SE SDK (e.g. version 25) includes:

- `javac` compiler
- `jdb` debugger
- JRE (Java Run Time Environment)
  - `java` JVM
  - Native packages (awt, swing, system, etc)

  - Docs, i.e., https://docs.oracle.com/en/java/javase/25

Usually the development is performed using an Integrated development environment (IDE).

To prepare a complete developmente environment the following must be installed:

- JDK (25.0)

  You can install any distribution, e.g.:

  - https://docs.aws.amazon.com/corretto/latest/corretto-25-ug/downloads-list.html
  - https://adoptium.net/en-GB/temurin/archive/?version=25
  - https://www.azul.com/downloads/?version=java-25-lts&package=jdk

  Documentation: https://docs.oracle.com/en/java/javase/25/docs/api/index.html

- Visual Studio Code

  - Available at: https://code.visualstudio.com
  - Install the *Extension Pack for Java* from within VSCode on the first execution

- Maven build tool

  - Available at: https://maven.apache.org/download.cgi
  - Instructions: https://maven.apache.org/install.html

- Git version control system

  - Available at: https://git-scm.com/downloads

4.3.1. Build and run

The build and run process is show in Figure 4.1:

- the source code contained in the `First.java` file is compiled by the Java compiler `javac` with the command

  ```
  javac First.java
  ```

  the output of the compilation is the `First.class` file that contains the bytecode for the class.

- the `main` method in the class can be executed using the JVM `java` that can be run with the command:

  ```
  java First
  ```

  note that the command has not `.class` extension. The JVM takes the name of the class and looks for the corresponding bytecode file by name.

- the JVM executes the code as with any other programming language.

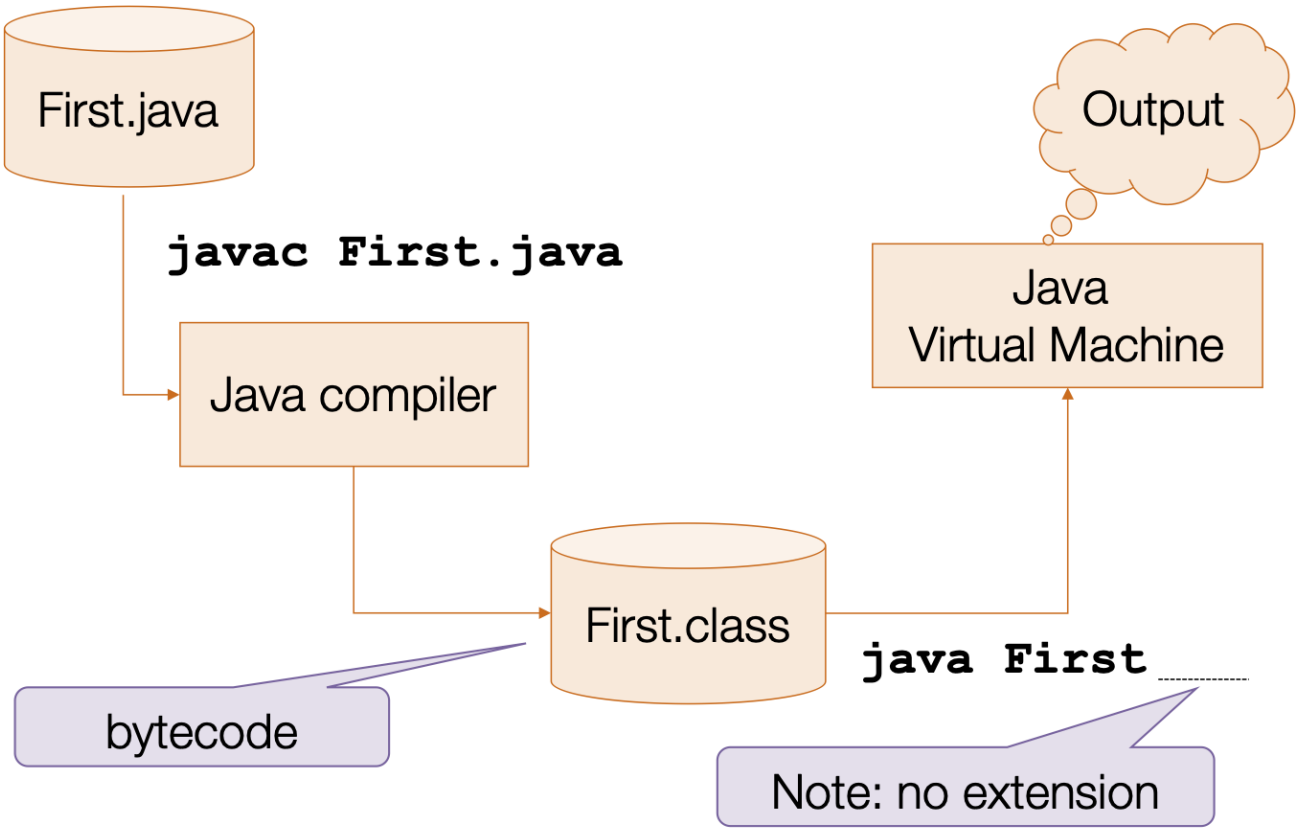


Figure 4.1.: Build and Run steps

The compiler `javac` takes a list of source files to be compiled and generates the corresponding `.class` bytecode files.

If a java file refers to a class that is not included in the list, the compiler looks in the predefined libraries and if not found looks for the source code file and adds it to the compilation list.

After all files have been compiled, the compiler checks for cross-reference integrity.

The key of Java platform portability lies in the bytecode. It is an executable intermediate code that can be run by the JVM. The bytecode is independent from the processor architecture or the operating system. As long as you have a JVM for your platform you can execute the bytecode, which is always the same.

Java bytecode `.class` files start with the following 4 bytes: `0xCAFEBABE`. This is called the *file magic number*; it is a specific initial sequence of a file that let the OS and other programs identify its content type, e.g. PDF files start with chars "`%PDF`"

JVM loading is based on the *classpath* that is the list of locations whence classes can be loaded.

When class `Cls` is required, the JVM, for each location in the classpath:

- Looks for file `Cls.class`
- If present, loads the class
- Otherwise moves to next location and repeats

Example file: `First.java`:

```
public class First {
  public static void main(String[] args){
    int a;
    a = 3;
      IO.println(a);
  }
}
```

The execution of the command: `java First` by the JVM entails the following steps:

- takes the name of the class (`First`), which means that the corresponding bytecode will be in a file named `Fist.class`,
- looks for the bytecode for that class in the classpath (and `.` eventually),
- loads the class's bytecode,
- performs all due initializations,
- looks for a suitable `main()` method in the class,
- start the execution by calling the `main()` method.

If the Java VM is not able to find a class it shows an errore exception of `ClassNotFoundException`. A typical reason for this error is that the name of the class on the command line must not include the extension `.class`; e.g.. `java First` not `java First.class`. Another possible cause is that the JVM looks for classes starting from the current working directory.

Since the JVM interprets the instructions that use the bytecode notation, it is slower that other languages that are compiled directly into the CPU native code.

The typicial questions: *Which is more "powerful": Java or C?* Can be asnwered with **it depends**:

- Performance-wise: C is better though non that much better. If an unexperience programmer starting from the official JVM specification execution times of equivalent programs can be expected to increased by 20X. The original JVM sported a time increase of around 5 to 10 times, this is not only due to the inherent overhead of the interpretation but also to the additional run-time checks performed by the JVM (e.g. array boundaries). Modern JVMs use a technique called Just-in-time compilation (JIT), using this approach the JVM takes the bytecode, translates it into the native code for the CPU, and calls the native code. Using this approach the resulting overhead can be aroung 10-15%.
- In term of Ease of use: Java is extremely easier to use
- As far as error containment is concerned, Java is far superior to what we can achieve in C thanks to more detailed checks at both compile time and run time (which are also a cause of the performance penalty).

4.3.2. Types of Java programs

- Application
  - It's a common program, similarly to C executable programs
  - Runs through the Java interpreter (java) of the installed JVM

```
public class HelloWorld {
    public static void main(String args[]){
        IO.println("Hello world!");
    }
}
```

- Servlet (web server)
  - In J2EE (Java 2 Enterprise Edition)
- Midlet (mobile devices)
  - In J2ME (Java 2 Micro Edition)
- Android App (Android device)
- Applet (client browser) now deprecated
  - Java code dynamically downloaded
  - Execution is limited by a browser "sandbox"

4.4. Deployment

In java there is no linking step that compines together several different object files into a single executable. All the bytecode `.class` files are independent dynamically loaded code fragments.

To simplify deployment, often Java programs are packaged and deployed in `.jar` files.

Jar files are compressed archives using the zip file format, that contain all the `.class` files required for an application. A `.jar` file contains additional meta-information that can be used for the deployment and execution.

The JVM is capable of accessing directly the `.class` files that are contained in a `.jar`, without the need to decompress it.

The SDK includes the `jar` program to perform all operation with `.jar` files. The main jar commands are:

- A jar file can be created using, e.g.,
  ```
  jar cvf my.jar *.class
  ```
- The contents can be seen with:
  ```
  jar tf my.jar
  ```
- To define a main class, a manifest file must be added to the jar with:
  ```
  jar cvfm my.jar manifest.txt
  ```
  Simplest manifest to define the main class:
  ```
  Main-Class: First.java
  ```

Usage of jar files:

- To run a class included in a jar:
  ```
  java -cp my.jar First
  ```
  The `-cp my.jar` option adds the jar to the JVM classpath
- When a main class for a jar is defined, it can be executed simply by:
  ```
  java -jar my.jar
  ```

In Java there is not the equivalente of a single stand-alone executable file such as an ".exe". Although using the GNU porting of the Java compiler (GCJ) is is possible to compile Java code into native code, combine it with the libraries and generate a standalone executable file.

4.5. Coding conventions

- Use `camelBackCapitalization` for compound names, not underscore
- Class name must be Capitalized
- Method names, object instance names, attributes, method variables must all start in lowercase
- Constants must be all uppercases (w/ underscore): `DEFAULT_VALUE`
- Indent properly

Example:

```
class ClassName {
    final static double PI = 3.14;                    ①

    private int attributeName;                        ②

    public void methodName() {                        ③
        int var;                                      ④
        if ( var==0 ) {
        // ...
        }
    }
}
```

① constants are capitalized, and possibly use _ to join words
② attributes start with lower case and are camel cased
③ method names start with lower case and are camel cased
④ variables start with lower case and are possibly camel cased

4.6. FAQ

- I downloaded Java on my PC but I cannot compile Java programs:
  - Check you downloaded Java SDK (including the compiler) not Java RTE or JRE (just the JVM)
  - Check the OS path includes the `path/to/java/bin`
  - Note: IDEs often use a different compiler than javac minor differences can be expected

4.7. Wrap-up

- Java is a quasi-pure OO language
- Java is interpreted
- Java is robust (no explicit pointers, static/dynamic checks, garbage collection)
- Java provides many utilities (data types, threads, networking, graphics)
- Java can used for different types of programs
- Coding conventions are not “just aesthetic”

5. Basic Features

Learning objectives

- Learn the syntax of the Java language
- Understand the primitive types
- Understand how classes are defined and objects used
- Understand how modularization and scoping work
- Understand how arrays work
- Learn about wrapper types

5.1. Fundamental syntax

5.1.1. Comments

Java allows both C-style comments (multi-lines):

```
/* this comment is so long
   that it needs two lines */
```

and comments on a single line:

```
  // comment on one line
```

5.1.2. Code blocks and Scope

Java code blocks are the same as in C

Each block is enclosed by *braces* `{ }` and starts a new *scope* for the variables

Variables can be declared both at the beginning and in the middle of a block before their use (differently from ANSI C):

```
for (int i=0; i<10; i++){
  int   x = 12;
  ...
  int y;
  ...
}
```

5.1.3. Control statements

- Similar to C
  - `if-else`
  - `switch`
  - `while`
  - `do-while`
  - `for`
  - `break`
  - `continue`

Switch statements can be used with strings:

```
switch(season){
  case "summer":
  case "spring": temp = "hot";
                 break;
  case "winter":
  case "fall": temp = "cold";
                 break;
}
```

Compiler generates more efficient bytecode from switch using `String` objects than from chained if-then-else statements.

5.1.4. Boolean

Java has an explicit type (`boolean`) to represent logical values (`true` , `false`)

Conditional constructs require boolean conditions. It is illegal to evaluate integer values as conditions.

```
int   x = 7;
if(  x  ){ ... }   //NO
```

In Java relational operators are mandatory to check if a value is (not) zero:

```
if (x != 0){ ... }
```

This avoids mistakes that are common in C, e.g.

```
if(x=0) ...
```

5.1.5. Passing parameters

Parameters are always passed *by value* they can be primitive types or object *references*. Only the object reference (pointer) is copied not the whole object.

5.1.6. Primitive Types

They are fefined in the language: `int`, `double`, `boolean`, etc.

The declaration of an instance of a primitive type:

- Declares the instance name
- Declares the type (name is associated to a specific type)
- Allocates memory space for the value

The primitive types available in the Java language are reported in Table 5.1.

Table 5.1.: Java primitive types

Type	Size	Encoding
`boolean`	1 bit	-
`char`	16 bits	Unicode UTF16
`byte`	8 bits	Signed integer 2C
`short`	16 bits	Signed integer 2C
`int`	32 bits	Signed integer 2C
`long`	64 bits	Signed integer 2C
`float`	32 bits	IEEE 754 sp
`double`	64 bits	IEEE 754 dp
`void`	-	

Concerning the boolean type it is important to remember that the logical size does not match the memory occupation.

5.1.7. Literals

- Literals of type `int`, `float`, `char`, strings follow C syntax
  - `123 256789L 0xff34 123.75 0.12375e+3`
  - `'a' '%' '\n' "prova" "prova\n"`
  - in addition since Java 15, also multiline strings (a-la python)

```
"""
...
"""
```

- Boolean literals (do not exist in C) are
  - **`true`** , **`false`**

5.1.8. Operators

Operators follow C syntax:

- arithmetical `+ - * / %`
- relational `== != > < >= <=`
- bitwise (int) `& | ^ << >> ~`
- Assignment `= += -= *= /= %= &= |= ^=`
- Increment `++ --`
- Logical `&& || ! ^`.

Chars are considered like integers (e.g. switch)

Logical operators work only on `boolean` operands.

Relational operators return `boolean` values.

5.2. Classes and Objects

The Java language defines structural elements (i.e., types) that are used and declared at compile time:

- Class
- Primitive type

The execution involves dynamic elements (i.e., instances) that are active at run time

- Reference
- Variable

Table 5.2.: Classes and primitive types

category	type	variable
type primitive	`int`	`int i;`
Class	`class Exam {}`	`Exam e;` `e = new Exam();`

5.2.1. Class

A class is defined by developer (e.g., `Exam`) or in the Java runtime libraries (e.g., `String`)

The declaration of a class type variable allocates memory for the *reference* ('pointer').

A reference variable (a variable whose type is a class) can be initialized to the special value `null` to indicate that it refers to no object. It is similar to the `NULL` value used for pointers in C. Class fields are initialized it with `null`, while local variables are considered **non initialized** and thus not usable.

Allocation and initialization of the *object* value are made later by its constructor

A class is an object descriptor, it defines the common structure of a set of objects. It consists of a set of *members*:

- Attributes
- Methods
- Constructors

Example of class definition

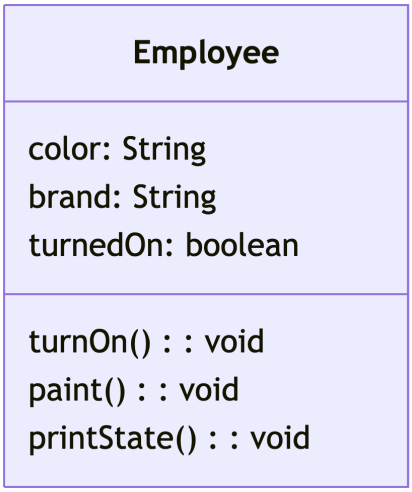
```
public class Car {
  // attributes
  String color ;
  String brand;
  boolean turnedOn;

  // methods
  void turnOn(){
    turnedOn = true;
  }

  void paint(String newCol) {
    color = newCol;
  }

  void printState () {
    IO.println ( " Car " + brand + " " + color );
    IO.println ( " the engine is "
                              +( turnedOn ? " on " : " off " ));
  }
}
```

The corresponding UML representation is:



Attributes (aka fields) describe the data that can be stored within objects. They are like variables, defined by:

- Type
- Name

Each object has its own copy of the attributes

Methods (aka operations) represent the messages that an object can accept, in the example above *turnOn*, *paint*, *printState*. Methods may accept arguments, e.g. `paint( String )`.

5.2.2. Object

An object is identified by:

- Class, which defines its structure (in terms of attributes and methods)
- *State* (values of attributes)
- *Internal* unique *identifier*

An object can be accessed through a reference. Any object can be pointed to by one or more references, this is called *aliasing*.

```
class ExLifeCycle(){
public static void main(String[] args){
  // declare reference
  Car c;

  // create object
  c = new Car();

  // use object
   c.paint( "yellow " );
}  // reference is lost upon method exit
}
```

Objects and references example

```
Car a1, a2;                    ①
a1 = new Car();                ②
a1.paint( "yellow" );          ③
a2 = a1;                       ④
a1 = null;                     ⑤
a2 = null;                     ⑥
```

① Two *uninitialized* references are created, they can't be used in any way. Rememeber: a reference is not an object just a smart pointer.
② An object is created and the "pointer" stored into the reference `a1`
③ Method `paint()` is invoked on the object through the reference `a1`

④ Two references point to the same object. This is known as *aliasing
⑤ Only one reference points to the object
⑥ No reference pointing to the object, which is *unreachable* and may be disposed of by the garbage collector

Objects Creation

Creation of an object is performed using the keyword **`new`**. It returns a reference to the area of memory containing the newly created object

```
Car m = new Car();
```

The keyword `new`:

- Creates a new instance of the specific class
- Allocates the required memory in the heap
- Calls the constructor of the object
- Returns a reference to the new object

5.2.3. Constructors

Constructor is a special method containing the operations (e.g. initialization of attributes) to be executed on each object as soon as it is created

It has **no return type** and is **named like the class**.

```
public class Car {
  boolean turnedOn;

  public Car(){
    turnedOn = false;
  }
}
```

Constructor may have parameters, e.g.

```
public class Car {
  String brand;

  public Car(String b){
    brand = b;
  }
}
```

This enables providing initial values for the state of the object on creation:

```
Car myCar = new Car("Ferrari");
```

Overloading of constructors is often used to make object creation more flexible.

If no constructor *at all* is declared, a default one with no arguments is provided implicitly by the compiler that does nothing.

Attributes are always initialized before the exceution of any possible constructor. Attributes are initialized with default values:

- Numeric: `0` or `0L` or `0.0` (zero)
- Boolean: `false`
- Reference: `null`

Note that the return type *must not* be declared for constructors. If a return type is inserted by mistake, the constructor is considered a method and it is not invoked upon instantiation! Usually IDEs signal this kind of mistake as a warning.

5.2.4. Current object – a.k.a. `this`

During the execution of a method it is possible to refer to the current object using the keyword `this`: the object upon which the method has been invoked.

In the methods this is used to refer to the current object and enable accessing attributes (and methods):

```
  void turnOn(){
    this.turnedOn = true;
  }
```

Note that `this` makes no sense within methods that are not invoked on an object, e.g. the `main` method.

5.2.5. Dotted notation

A method can be invoked using dotted notation

```
objectReference . method( parameters ... )
```

Example:

```
Car a = new Car();
a . turnOn();
a . paint( "Blue" );
```

Note:

If a method is invoked from within another method of the *same object* dotted notation is not mandatory, in such cases `this` is implied.

```
class Book {
  int pages;
  void readPage(int n) { … }
  void readAll  () {
    for(int i=0; i <pages; i++){
      readPage(  i  ); // equivalent to this.readPage(i)
    }
  }
}
```

It is not mandatory and it is automatically added by the compiler:

```
class Book {
  int pages;
  void readPage(int n) { … }
  void   readAll  () {
    for(int   i  =0;   i  <pages;   i  ++){
      this . readPage(  i  );
    }
  }
}
```

Access to attributes is perfomed similarly, using the dotted notation

```
objectReference . attribute
```

A reference is used like a normal variable

```
Car a = new Car();
a  . color  = "Blue"  ;  //what's wrong here?
boolean x = a.turnedOn  ;
```

Methods accessing attributes of the *same object* they were invoked upon do not need to use the object reference

```
class Car {
  String color;
  …
  void paint(){
    color = "green";  //  * color refers to current obj
  }
}
```

Athough the use of `this` is not mandatory, it can be useful in methods to disambiguate object attributes from local variables and arguments

```
class Car{
  String color;
  ...
  void paint(String color) {
    this.color = color;
  }
}
```

The explicit use of `this` for methods and attributes, is not dicated by any universal Java code convention. It is more a matter of personal taste. Nevertheless, conventions at team, project, or organization level might mandate or discourage it.

5.2.6. Chaining dotted notations

Multiple dotted notations can be combined in a single expression

Method chaining is a style of designing the methods of a class so that the continuous invocations of several methods can be expressed with a single statement instead of multiple separate statements.

```
public class Counter {
    private   int   value;
    public Counter reset(){
        value=0;
    return this;
    }
    public Counter increment(  int   by){
        this.value  +=by;
    return this;
    }

    public Counter print(){
        IO.println  (value);
        return this;
    }
}
```

If we want to perform multiple operations on a counter, the conventional approach is to have multiple statements:

```
Counter cnt = new Counter();
cnt.reset();
cnt.print();
cnt.increment(10);
cnt.print();
```

```
cnt.decrement(7);
cnt.print();
```

Since all the methods return `this`, it is possible to use an alternative compact, single statement, expression:

```
Counter cnt = new Counter();
cnt.reset().print()
   .increment(10).print()
   .decrement(7 ).print();
```

A very common example of chained notation is the user of `println()`:

```
System . out . println("Hello world!");
```

- `System` is a class in package `java.lang`
- `out` is a (static) attribute of `System` referencing an object of type `PrintStream` (representing the standard output)
- `println()` is a method of `PrintStream` which prints a text line followed by a new-line

5.2.7. Operations on references

The main operator that can be applied to references is the `.` (dot) that is usede to de-referentiate the reference and access the corresponding object.

There is *NO* pointer arithmetic on references in Java. Only the comparison operators `==` and `!=` are defined

The relational operators check whether two references points to the same object (or not). They check identity, but do not check on the objects' contents.

To compare contents an ad-hoc method – e.g. `equals()` – must be defined, which defines the criteria to compare the contents (values of attributes) of two objects.

For instance we could compare two cars based on their color:

```
class Car{
  String color;
  ...
  boolean equalsColor(Car other) {
    return this.color.equals(other.color);                    ①
  }
}
```

① the method `equals()` of `String` checks the content of the strings

Then it it possible to compare two cars based on their contents:

```
Car a = new Car();
a.paint("red");
Car b = new Car();
b.paint("red");

sameObject = a == b;                                                    ①
sameColor = a.equalColor(b);                                            ②
```

① `a` and `b` are evidently not the same object
② but they have the same color

5.2.8. Overloading

Several methods in a class can share the same name. They must have have distinct *signature*.

In Java, the signature of a method consists of:

- the method name
- the ordered list of argument types

The invocation of an overloaded method is potentially ambiguous. Disambiguation is performed by the compiler based on actual parameters. The method definition whose argument types list matches the actual parameters, is selected.

Overloading example:

```
class Car {
  String color;
  void paint(){
    color = "white";
  }

  void paint(int i){
    switch(i){
      case 0: color = "white"; break
      case 1: color = "red"; break
      case 2: color = "green"; break
      case 3: color = "blue"; break
      default: color = "black"; break
    }
  }

  void paint(String newCol){
    color =   newCol  ;
  }
}
```

The compiler decides which variant of an overload to invoke based on the arguments, starting from the first and going on until a unique method is identified, other arguments are then converted (if possible)

```
public class Foo{
  public void doIt(int x, long c){ IO.println("a"); }
  public void doIt(long x, int c){ IO.println("b"); }
  public static void main(String args[]){
    Foo f = new Foo();
    f.doIt(5, (long)7); // a
    f.doIt((long)5, 7); // b
  }
}
```

Constructors with overloading

```
class Car {
  // …
  //    Default constructor, creates a red Ferrari
  public Car(){
        color = "red";
        brand = "Ferrari";
    }

  // Constructor accepting the brand only
  public Car(String carBrand){
        color = "white";
      brand = carBrand;
    }

  // Constructor accepting the brand and the color
    public Car(String carBrand, String carColor){
        color = carColor;
        brand = carBrand;
    }
}
```

5.3. Scope and encapsulation

5.3.1. Visibility modifiers

Visibility modifiers are applicable to members of a class

- `private`
  - Member is visible and accessible from instances of the same class only
- `public`

- Member is visible and accessible from everywhere

With public attributes

```
class Car {
  public String color;
}
```

```
Car a = new Car();
a.color = "white"; // ok
```

With private attributes

```
class Car {
  private String color;
  public void paint(String color)
  { this.color = color;}
}
```

```
Car a = new Car();
a.color = "white";  // error
a.paint( "green");  // ok
```

The normal approach in terms of visibility is to make all attributes *private* by default. This is the application of the OO principle called **information hiding**. All the information managed by an object should be hidden from other objects and classes to avoid possible interference.

Table 5.3.: Basic visibility access rules

code in ↓ access to →	Private(attribute / method)	Public(attribute / method)
Method in the same class	yes	yes
Method in another class	no	yes

5.3.2. Getters and setters

Getters and setteres are the methods used to respectively read and write private attributes.

They allow to better control in a single point each write access to a private field

```
public String getColor() {
    return color;
}
public void setColor(String newColor) {
    color = newColor;
}
```

Example without getter/setter

```
public class Student {
  public String first;
  public String last;
  public int id;
  public int numExams;
  public Student(…){…}
}
```

```
public class Exam {
  public int grade;
  public Student student;
  public Exam(…){…}
}
```

```
class StudentExample{
    public static void main(String[]   args  ) {
    // defines a student and her exams
    // lists all student's exams
    Student s=new Student("Alice", "Green", 1234);
    Exam e = new Exam(30);
    e.student = s;
    s.numExams++;
    // print vote
    IO.println(e.grade);
    // print student
    IO.println(e.student.last);
    }
}
```

With getter and setters and delegation

```
class StudentExample{
    public static void main(String[]   args  ) {
    // defines a student and her exams
    // lists all student's exams
    Student s=new Student("Alice", "Green", 1234);
    Exam e = new Exam(30);
    e.setStudent(s);
    // prints its values and asks students to
    // print their data
    e.print();
    }
}
```

```
public class Student {
    private String first;
    private String last;
    private int id;
  private numExams=0;
    public String toString() {
        return first + " " +
            last  + " " +
            id;
    }
}
```

```
public class Exam {
    private int grade;
    private Student student;
      public void print() {
          IO.println( "Student " + student.toString() +
                          " got " + grade);
      }

      public void setStudent(Student s) {
          this.student = s;
      s.numExams++;
    }
}
```

Getters and setters vs. public fields

- Getter
  - Allow changing the internal representation without affecting
    - E.g. can perform type conversion
- Setter
  - Allow performing checks before modifying the attribute
    - E.g. Validity of values, authorization
  - Allow side effects to keep data consistent
    - E.g. Updatinge related information

5.3.3. Modifier / Query methods

In general it is possible to classify methods into two main categories:

- **Modifiers**: change the state of the object but do not return a value, e.g. setters
- **Query**: return a result and do not change the state of the object, they have no side-effects, e.g. getters

In general it is possible to have a method that both has side effects and return a value.

A principle of good desing is the **Modifier / Query Separation**:

> a method should be either a modifier or a query.

In practice to separate them:

- Queries return a value
- Modifiers return `void`

The consequence of this principle applies to the code using the conforming classes:

- invocations to
  - queries can be added, removed, and swapped without affecting the overall behavior
  - modifiers cannot be touched without affecting the behavior

The idea si described by Martin Fowler here https://www.martinfowler.com/bliki/CommandQuerySeparation.html. The original concept was proposed by Meyer (1997).

5.4. Package

The basic unit of Object-Oriented langauges, the *Class* is a better mechanism of modularization than a procedure. Although it is still small, when compared to the size of large applications.

For the purpose of code organization and structuring Java provides the **package**. A package is a *logic set* of class definitions. These classes consist in several files, all stored in the *same folder* Each package defines a new *scope* (i.e., it puts bounds to visibility of names) It is therefore possible to use *same class names in different package* without name-conflicts

A package is identified by a name with a hierarchic structure (*fully qualified name*), e.g. `java.lang` contais basic language classes such as `String`, `System`, etc.

The universally applied convention to create unique names for packages is to use internet names in reverse order. For instance packages developed at Politecnico di Torino (polito.it) will be nameed `it.polito.mypackage`.

Examples of predefined packages

- `java.awt` contain classes
  - `Button`
  - `Window`
  - `Menu`
  - etc.
- `java.awt.event` is a sub-package containing classes:
  - `MouseEvent`
  - `KeyEvent`
  - etc.

5.4.1. Creation and usage

Declaration of a packages is done through the package statement at the beginning of each class file:

```
package it.polito.packagename;
```

Classe in other packages are not automatically accessible, they need to be imported.

In the initial part of the file – before the class declaration – import statements declare which classes will be accessible:

```
import packageName.className;
```

In addition to importing a single class for each `import` statement, it is possible to import all the classes in a package:

```
import packageName.*;
```

If there is not an import statement, to access a class in a specific package the fully-qualified name of the classe is required, that is the name of the package and the name of the class.

```
int i = myPackage.Console.readInt()
```

The possibility to use the fully-qualified name permits using two classes with the same name that are defined in different classes. Two `import` statements for classes with the same name is an error, because it creates a name clash. Althouht is is possible to import one and directly refer to the other.

```
import   java.sql.Date  ;
\\...
Date d1; // java.sql.Date
\\...
java.util.Date d2 = new java.util.Date();
```

The need to import classes residing in a different packages does not apply to the special package `java.lang` that contains a few fundamental classes, e.g. `String` and `System`. The compiler automatically imports all classes in the `java.lang` package as if there were an initial:

```
import java.lang.*
```

5.4.2. Default package

When no package is specified, the class belongs to the so-called **default package**.

The default package has no name. As a consequence classes in the default package cannot be accessed by classes residing in other packages. For this reason, the usage of default package is considered a bad design practice and it is discouraged.

5.4.3. Package and scope

There are specific scope and visivility rules that apply to packages.

The "*interface*" of a package is the set of *public classes* contained in the package. A package can be considered as an entity of modularization. The principle of information hiding can be applied also at package scale. The idea is to minimize the number of classes, attributes, methods visible outside the package.

To take into consideration the package level scope, a third level of visibili **package visibility** is available in Java. It is represented by the *absence of any modifier*, i.e. element without either a `public` or `private` are considered package only visible. Example:

- `public class A { }`
  - Class and public members of A are visible from outside the package
- `class B { }`
  - Class and any members of B are not visible from outside the package
- `private class C { }`
  - Illegal: a private class is not accessible from any other class, therefore the class and its members would be visible to themselves only

Multiple packages

```
package it.polito.sample.one;

public class A {
  public int a1;
  private int a2;
  int a3;
  public void m1(){}
}
```

```
package it.polito.sample.one;

class B {
  public int b1;
  private int b2;
  int b3;
}
```

```
package it.polito.sample.two;

class C {
  public void m2(){}
}
```

The code in method `m1()` of class `A` can access all attributes (`a1`, `a2`, `a3`) in the same class, it can also access `b1` in class `B`, cannot access `b2` since it is private, but it can access b3 since it is package visible.

The code in method `m2()` of class `C` can access `a1` in class `A` but neither `a2` nor `a3`, it cannot access any member in class `B` since the class itself is pacakge visible and `C` resides in a different package.

Table 5.4.: Summary of access rules

code in ↓ access to →	Private(attribute / method)	Package(attribute/method)in public class	Public(attribute/method)in package class	Public(attribute/method)in public class
same class	yes	yes	yes	yes
other class same package	no	yes	yes	yes
other class other pacakge	no	no	no	yes

5.5. Wrapper Classes

In an ideal OO world, there are only classes and objects, although, for the sake of efficiency, Java – as well as other OO languages – uses primitive types (`int`, `float`, etc.).

Wrapper classes are object versions of the primitive types, they encapsulate a single attribute of the corresponding primitive type.

They are defined in the `java.lang` package.

Table 5.5.: Wrapper classes in Java

Primitive type	Wrapper Class
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`void`	`Void`

5.5.1. Conversions

The wrapper classes define a few method that carry on *conversion operations* between different types

An example of conversions for the `int`, `Integer` and `String` type is reported in Table 5.6. Similar conversions are available for all other numeric types.

Table 5.6.: Integer conversions

V from to ->	`int`	`Integer`	`String`
`int`		`Integer.valueOf(i)`	`String.valueOf(i)`
`Integer`	`I.intValue()`		`I.toString()`
`String`	`Integer.parseInt(s)`	`Integer.valueOf(s)`	

Given an `int` value, the corresponding `Integer` object can be build in two different ways:

- using a constructor `new Integer(42)`, that is discouraged because it is not efficient (see [Pooling][#pooling]),
- using the `valueOf()` method, which is the recommended methdo

Example:

```
Integer obj = Integer.valueOf(88);
String s = obj.toString();
int i = obj.intValue  ();
int j = Integer.parseInt("99");
int k = (obj).intValue();
```

5.5.2. Autoboxing

Since Java 5, the conversion between primitive types and wrapper classes is performed automatically (*autoboxing*)

```
Integer i = Integer.valueOf(2);
int j;
j =   i   + 5;
    _//instead of: _
j =   i   .intValue ()+5;

i   = j + 2;
      _//instead of:_
i   =    Integer.valueOf (j+2);
```

5.5.3. String

There is no primitive type to represent string

String literal is a quoted text

In C:

- `char s[] =  "literal"`
- Equivalence between strings and char arrays

In Java

- `char[]` != `String`
- *`String` class* in `java.lang` package

See Java Characters and Strings

5.6. Arrays

An array is an *ordered sequence* of variables of the same type which are accessed through an *index*

In Java an array is an *object* and it is created with `new` and stored in the heap.

An array can contain either *primitive types* or *object references* (not object values).

Array *dimension* can be defined at run-time, during object creation, and cannot be change afterwards. That is arrays in Java are not resizable.

5.6.1. Declaration and Creation

An array reference can be *declared* with one of these equivalent syntaxes

```
int[] a;                                                                    ①
int a[];                                                                    ②
```

① Java style recommended
② C style, accepted but not recommended

The array declaration allocates memory space for a *reference* to the array, it does not allocates any memory at all for the array itself, i.e. the elements of the array are not created with the simple declaration.

The array object (i.e. the elements of the array) can be created using either:

- the **`new`** operator, initialize all the elements to `0` or `null`, or
- a *static initialization* , providing the array initial elements values.

Example:

```
int[] a;                                                                    ①

a = new int[10];                                                            ②

String[] s = new String[5];                                                 ③

int[] primes = {2, 3, 5, 7, 11, 13};                                        ④

String[] p = { "John", "Susan" };                                           ⑤
```

① declaration of an array of `int`
② creation of an array of 10 `int` elements, all the element are initialized to `0`
③ declaration and creation of an array of 5 `String` (references), all the elements is initialized to `null`
④ declaration and static initialization of an array with 6 `int` elements whose values are listed,
⑤ declaration and static inizialization of an array with 2 `String` references initialized to the given strings.

5.6.2. Operations on arrays

Elements are selected with brackets `[ ]` (C-like), but Java makes bounds checking at run-time. Array length (number of elements) is given by attribute `length`

Any access to an invalid inded for an array results in an error. In particular an `ArrayIndexOutOfBoundsException` exception is produced. The program usually terminates with an error message that indicates the offending index and the length of the array, e.g., `Index 8 out of bounds for length 8`.

The common idiom for iterating on the elements of an array in Java is:

```
for(int i=0; i < a.length; i++){
  a[i] = i;
}
```

It is important to remember that an array reference is *not* a pointer to the first element of the array. It is a *pointer* – a reference indeed – to the array *object*, that contains the elements. It is importanto to remember that arithmetic on pointers does not exist in Java and definitely does not work with arrays.

Since Java 5 a new loop construct, called *for-each* is available to iterate on arrays and other containers It is a more compact notation with respect to the default idiom:

`for( Type var : set_expression)`

Where the `set_expression` can be either an array a class implementing `Iterable`

When the *for-each* construct is used, the compiler can generate automatically the loop with correct indexes. In general the usage of this construct is recommended since it is less error prone.

Example:

```
for(String arg : args ){
  IO.println(arg);
}
```

is equivalent to

```
for(int i=0; i < args.length; ++i){
    String arg = args[ i ];
  IO.println(arg);
}
```

5.6.3. Multidimensional arrays

In Java, multidimensional arrays are implemented as array of arrays:

```
Person[][] table = new Person[2][3];
table[0][2] = new Person("Mary");
```

Since rows are not stored in adjacent positions in memory they can be *easily changed*

```
double[][] balance = new double[5][6];
\\...
double[] temp = balance[i];
balance[i] = balance[j];
balance[j] = temp;
```

It is also possible to have rows with different lengths since the type of an array the same independently of its length.

5.6.4. Variable arguments

It is possible to pass a variable number of arguments to a method using the *varargs* notation

```
method( type ... args )
```

The compiler assembles an array that can be used to scan the actual arguments. Type can be primitive or class.

Example

```
static int min(int ... values){
        int res = Integer.MAX_VALUE;
        for(int v : values){
      res=v<res?v:res;
        }
        return res;
}

public static void main(String[] args) {
  int   m = min(9,3,5,7,2,8);
  IO.println("min =" + m);
}
```

5.7. `static` and `final` modifiers

The `static` modifier can be applied to both attributes and methods to make them class-level members as opposed to the regular (non static) instance members.

5.7.1. Static attributes

A *static* attribute represents a property which is common to all instances of a class A single copy of a *static* attribute is shared by all instances of the class. Sometimes they are called *class attributes* as opposed to the regular *instance attributes*.

Static attributes of a class exist before any object of that class is created. A change performed by any object is visible to all instances at once.

Static attributes can be used to keep a shared property, e.g. a count of created instances, or a pool of all instances, or to keep a common constant value used by all objects.

```
class Car {
  static int countBuiltCars = 0;

  public Car(){
    countBuiltCars++;
  }
}
```

> **Note**
>
> The use of static attributes should be limited to few very limited a specific cases. In general it is a good programming practice to **avoid `static` attributes** because they can make the code untestable and hard to debug.

5.7.2. Static methods

Static methods are not related to any instance, that is they do not define the implicit `this` reference. Therefore they cannot direcly access a regular (instance) attribute without explicitly dereferencing a reference.

The advantage of static methods is that they do not need an object to be created in order to invoked and executed. This is particularly useful, e.g., for the `main` method, which is called by the JVM when a program is started.

```
public class HelloWorld, {
  public static void main (String args[]) {
     IO.println("Hello Java World!");
  }
}
```

If the `main` method is not declared as `static` the JVM need to instantiate an object (if a suitable constructor is available) and then invoke the method.

```
class Car {
  private static int countBuiltCars = 0;

  public Car(){
    countBuiltCars++;
  }

  public static int howManyCars(){
    return countBuiltCars;
  }
}
```

To access a static member – both attribute and method – the name of the class is used:

```
Car.howManyCars(10);
```

It is possible to import all static items using the `import static` statement

```
import static pakage.Utility.*;
```

Such statment makes all static members of the imported class accessible with their simple name, without the need to prefix them with the class name.

There are two main motivations for using static methods:

- Implement *functions*
  - methods whose result depends exclusively on the arguments, avoid creating an object just to invoke the method (see e.g., `main()`).
- Provide ideal *factory method*
  - method that can be used to create an instance

5.7.3. Function methods

A "*function*" is a method whose return value depends only on the arguments. Functions are typically defined as `static` because they do not need any attribute.

Such functions are often collected within a *utility* class, that is a class containing `static` function methods only.

Wrapper types include several *function* methods for conversion purposes, although they are not utility classes.

There are several examples, in the Java standard library, of predefined utility classes:

- `Math`
  - Mathematical functions

- `Arrays`
  - Functions to operate on arrays
- `IO`
  - Functions to perform console I/O
- `System`
  - Interact with the operating system
- `Objects`
  - Functions to operate on object

5.7.3.1. Class Math

Defines several math-related function methods:

- Trigonometric functions (`sin()`, `cos()`, `tan()`)
- Min-max (`min()`, `max()`)
- Exponential and logarithms (`exp()`, `log()`)
- Truncations (`round()`, `ceil()`, `floor()`)
- Random number generation (`random()`)
- Powers (`power()`, `sqrt()`)

5.7.3.2. Class Arrays

Providea set of utility functions that work with arrays

- Binary search (`binarySearch()`)
- Copy (`copyOf()` , `copyOfRange()`)
- Equality (`equals()` , `deepEquals()`)
- Fill-in (`fill()`)
- Sorting (`sort()`)
- String representation (`toString()`)

5.7.3.3. Class IO

The class `IO` provides a few functions to perform console-based I/O operations. It defines the following methods:

- `static void print(Object obj)` Prints an object value (also primitives)
- `static void println(Object obj)` Prints an object value (also primitives) with newline
- `static void println()` Prints a new line
- `static String readln(String prompt)` Reads a string from the console with a prompt
- `static String readln()` Reads a string from the console

5.7.3.4. Class System

Class `System` is a general purpose utility class. The main functions are:

- `long currentTimeMillis()` Current system time in milliseconds
- `void exit(int code)` Terminates the execution of the JVM
- `final PrintStream out` Standard output stream,
  - Also `err` for standard error

5.7.4. Factory method

A *factory method* is a method used to create an object. It Encapsulates an explicit object creation with the `new` operator.

A factory method can serve several purposes:

- Perform a check on the parameter, in case the parameter are invalid it can retun, e.g. `null`, a constructor is not allowed to not return a value.
- Return objects from a pool to reduce the creation time and minimize memory occupation of immutable objects
- Maintain a collection of created objects for any purpose
- Control the allocation of new objectsm, see e.g., Singleton pattern

An example of factory methods that create an `Integer` object are:

- `valueOf(int)`
  - Replaces new `Integer(int)`
  - Cache values in the range -128 to 127
- `valueOf(String)`
  - Returns the integer corresponding to the parsed string
  - Same as: `new Integer(Integer.parseInt(s))`

5.7.5. Final

When a attribute is declared as **`final`**, it cannot be changed after object construction. It can be initialized inline or by the constructor

```
class Student {
  final int years=3;
  final String id;
  public Student(String id){
      this.id   = id;
  }
}
```

When a method argument is declared as `final` it cannot be changed. Non final parameters are treated as local variables (initialized by the caller).

When a local variable is declared as `final` it cannot be changed after initialization. Note that initialization can occur at declaration or later

5.7.6. Constants

In Java the declaration of constants uses the combined `static final` modifiers. The `final` implies not modifiable, while `static` implies there is only one copy, i.e. it is non redundant.

```
final static float PI = 3.14;
…
PI = 16.0;         _// ERROR, no changes_
final static int SIZE;   // missing init
```

The conding convention in Java states that constants must be named all uppercase.

5.7.7. Static initialization block

A block of code preceded by `static` is called static initialization block. It can appear within class declaration, outside any method. It is executed at class loading time and can be used to initialize constants

```
public final static double 2PI;
static {
  2PI =   Math.acos  (-1);
}
```

5.7.8. Example: Global directory

Code that manages a global name directory

```
class Directory {
  public final static Directory root;
  static {
    root = new Directory();
  }
  // …
}
```

What if not always useful and expensive creation?

Manages a global directory

```
class Directory {
  private static Directory root;
  public static Directory   getInstance  (){
    if(root==null){
      root = new Directory();
    }
    return root;
  }
  // …
}
```

Created on-demand at first usage

5.7.9. Singleton Pattern

- Context:
  - A class represents a concept that requires a single instance
- Problem:
  - Clients could use this class in an inappropriate way

See slide deck on design patterns

PATTER IMAGE

```
public class Singleton{
  private Singleton() { } // cannot be instantiated using new
  private static Singleton instance;
  public static Singleton getInstance(){
    if(instance==null){
          instance = new Singleton();
    }
    return instance;
  }
}
```

5.7.10. Fluent Interfaces

Method to design OO API based on extensive use of method chaining

The goal is to improve readability

- Code looks like prose
- Often used to build complex objects

Creates a sort of Domain Specific Language (DSL) leveraging the syntax of the host language

See: https://www.martinfowler.com/bliki/FluentInterface.html

Example of usual, non-fluent, API to manage measure with the relative units:

```
Measure power = new Measure(10.4);
power.addUnit  ("kg", 1);
power.addUnit  ("m", 2);
power.addUnit  ("s", -3);
power.setPrecision  (2);
```

Fluent

```
Measure power = Measure.value(10.4).
        is("kg").by("m").squared().by("s").to(-3).
       withPrecision(2).done();
```

Can be implemented as

```
public class Measure {
  private double value  ;
  private Unit unit  ;
  private int precision  ;

  publicMeasure(double value) {
      this.value = value  ;
  }

  public void setPrecision(int precision) {
      this.precision = precision;
  }

  public void addUnit(String name, double exp){
      unit = new Unit(name, exp);
  }

  public static Builder value(double v){
      return new Builder(v);
  }
}
```

Fluent Builder

```
public static class Builder{
  private Measure object;
  private String unitName;
```

```
  public Builder(double v){ object = new Measure(v);}

  public Builder is(String name) {
      unitName = name;
      return this  ;
  }

  public Builder by(String name) {
    if(unitName != null) {
      object.addUnit( unitName, 1);
    }
    unitName = name;
    return this  ;
  }

  public Builder squared() {
      object.addUnit(unitName, 2);
      unitName = null;
      return this  ;
  }

  public Builder to(double exponent) {
      object.addUnit(unitName, exponent);
      unitName = null;
      return this  ;
  }

  public Measure done() { return object; }

  public Builder withPrecision(int precision) {
      object.setPrecision( precision );
      return this;
  }
}
```

5.8. Other Types

In addition to classes, that represent the main construct in Java, it is possible to define other types:

- enumerative types
- records
- interfaces

5.8.1. Enum

Defines an enumerative type

```
public  enum Suits {
  SPADES, HEARTS, DIAMONDS, CLUBS
}
```

Variables of enum types can assume only one of the enumerated values, e.g. `Suits card = Suits.HEARTS;` They allow much stricter static checking compared to integer constants (e.g. in C)

Enum can are similar to a class that automatically instantiates the values

```
class Suits {
   public static final Suits HEARTS=
                            new   Suits   ("HEARTS", 0);
   public static final Suits   DIAMONDS=
                                          new   Suits  ("DIAMONDS", 1);
   public static final Suits   CLUBS=
                                          new   Suits   ("CLUBS", 2);
   public static final Suits   SPADES=
                                            new   Suits   ("SPADES", 3);
   private Suits(String enumName, int index) {…}
 }
```

5.8.2. Record

The record keyword allow defining an immutable class with a compact notation. For instace to declare a 2D point:

```
record Point(int x, int y){}
```

The compiler transparenlty provides

- `public final` attributes for each parameter of the constructor,
- getter methods with the same name of the attributes, i.e. `x()` and `y()`.

It is also possible to define additional (query) methods.

Overall it is equivalent to a class but much more compact, the full class declaration would be:

```
class Point(){
  public final int x;
  public final int y;
  public Point(int x, int y){
    this.x=x;
      this.y=y;
  }
}
```

5.8.3. Interfaces

Interfaces are incomplete types that can define methods without implementation. They are mean to be implemented trough inheritance.

5.9. Nested Classes

It is possible to declare classes nested within other classes.

Java has several types of nested classes:

- Static nested class, the class is defined within the container name space and scope,
- Inner class, as the static and in addition the nested class contains a link to the creator container object
- Local inner class, defined in a method, similar to the inner class and in addition can access local variables
- Anonymous inner class, similar to the previous but has no explicit name.

5.9.1. (Static) Nested class

A class declared inside another class:

```
package pkg;
class Outer {
  static class Nested {
  }
}
```

Similar to regular classes, but is is subject to the member visibility rules. A nested class lies within the scope of the outer class. The fully qualified name includes the outer class: `pkg.Outer.Nested`

Static nested classes can be used to hide classes that are used only within another class, the advantages are:

- Reduce namespace pollution
- Encapsulate internal details

Example:

```
public class StackOfInt{

  private static class Element {                                    (1)
    int value;
    Element next;
    Element(int v, Element n){
      value=v;
      next=n;
```

```
    }
  }

  private Element head;                                                    ②

  public void push(int v){
    head = new Element(v,head);
  }

  public int pop(){
    int v = head.value;
    head = head.next;
    return v;
  }
}
```

① nested class that is visible only to members of the outer class `StackOfInt`
② the nested class `Element` is used as any other class

5.9.2. Inner Class

It is declared like a normal nested class but without the `static` modifier:

```
package pkg;
class Outer {
  class Inner{
  }
}
```

Any inner class instance is associated with the instance of its enclosing class that instantiated it (i.e. it is not static). As a consequence, it cannot be instantiated from a static method or from other classes. It has direct access to the enclosing object's methods and fields.

```
public class StackOfInt{

  private static class Element {                                           ①
    int value;
    Element next;
    Element(int v){
      value = v;
      next = head;                                                         ②
    }
  }

  private Element head;                                                    ③
```

```
  public void push(int v){
    head = new Element(v);                                    ④
  }

  public int pop(){
    int v = head.value;
    head = head.next;
    return v;
  }
}
```

① nested class that is visible only to members of the outer class `StackOfInt`
② the inner class `Element` can access the container object attribute `head`
③ the nested class `Element` is used as any other class
④ the `head` argument can be omitted since the inner class has direct access.

5.9.3. Local Inner Class

A local inner class is declared inside a method

```
public void m(){
  int   j=1;
  class X {
    int plus(){ return j + 1; }
  }
  X x = new X();
  IO.println  (  x.plus  ());
}
```

References to local variables are allowed, although they are replaced with "current" value. The set of such local variables is called *closure*

```
public void m(){
  int   j=1;
  class X {
    int plus(){ return j + 1; }
  }
  j++; // ERROR!
  X x = new X();
  IO.println(  x.plus() );
}
```

Local variable cannot be changed after being referred to by an inner class. Otherwise there would be ambiguity as to what result should we expect.

Local variables used in local inner classes should be declared *final*, or *be effectively* final.

5.9.4. Anonymous Inner Class

Local class without a name. Only possible with inheritance, i.e. implement an interface, or extend a class

See: inheritance in general and inheritance for further details.

5.10. Memory Management

5.10.1. Memory types

Depending on the kind of elements they include, Java manages three different types of memory:

- Static memory
  - elements living for all the execution of a program (class definitions, static variables)
- Heap (dynamic memory)
  - elements created at run-time (with 'new')
- Stack
  - elements defined in a code block (local variables and method parameters)

Heap memory is a part of the memory used by a program during execution to store data dynamically created at run-time. In Java class instances (objects) are always stored in the heap. The operator `new` returns a reference that points to a memory area in the heap.

In summary, Java has the following types of variables:

- Instance variables, also known as fields or attributes: stored within objects (in the heap)
- Local Variables: stored in the Stack; method pararameters are treated as local variables
- Static Variables: stored in static memory

5.10.2. Garbage collector

Memory release, in Java, is no longer a programmer's concern, Java is a *managed memory* language.

A component of the JVM called *Garbage Collector* cleans the heap memory from *dead* objects. Periodically it analyzes references and objects in memory and then it releases the memory for objects with no active references. There is no predefined timing when such – computationally intensive – procedure is performed. The method `System.gc()` can be used to *suggest* the GC to run as soon as possible, but the standard specification of Java does not mandate the suggestion will be obeyed.

5.10.3. Finalization

Before the object is actualy destroyed, i.e. the memory released, the method `public void finalize()` – if defined – is invoked. The `finalize()` method is intended to release resources owned by the object.

Warning: there is no guarantee an object will be ever released, therefore there is no guarantee that `finalize()` will be eventually called before program termination.

Finalization and garbage collection

```
class   Item {
  public   void   finalize(){
    IO.println("Finalizing");
  }
}
```java

```java
public static void  main(String args[]){
  Item i = new Item();
  i = null;
  System.gc(); //   _probably will finalize object_
}
```

5.11. Wrap-up

- Java syntax is very similar to that of C
- New primitive type: boolean
- Objects are accessed through references
  - References are disguised pointers!
- Reference definition and object creation are separate operations
- Different scopes and visibility levels
- Arrays are objects
- Wrapper types encapsulate primitive types

6. Characters and Strings

Character and string literals follow the C syntax with special chars quote with a \ backslash:

- Chars: `'a' '%' '\n'`
- Strings `"test" "line with new-line\n"`

```
"""
Multi line

text block
"""
```

Multiline text blocks have been introduced since Java 15.

6.1. Characters

Wrapper class `Character` encapsulates a single character. It is immutable like all wrapper classes. It provides a set of utility methods for characters:

- `isLetter()`
- `isDigit()`
- `isSpaceChar()`
- `toUpperCase()`
- `toLowerCase()`

6.1.1. Character sets

A character set is a set of characters in terms of abstract idea (e.g., the lowercase letter i of the latin alphabet) and the corresponding bit (and byte sequence) representation (e.g. $\texttt{01111001}_2$ or `0x69`).

6.1.1.1. ASCII

The ASCII (American Standard Code for Information Interchange) is the most widely adopted character set. The original version uses a mapping on 7 bits and is also coded in ECMA-6 (“7-bit coded character set” 1991).

It maps as set of commong symbols, the numerical digits, and the latin uppercase and lowercase letters.

It is often used in an extended format based on 8 bits and standardize by ISO/IEC 646 and ECMA-94 ("8-bit single-byte coded graphic character sets - Latin alphabets No. 1 to No. 4" 1986). The 8 bits versions include mainly letters with accents and additional symbols.

6.1.1.2. Unicode

Unicode("Unicode Specification," n.d.) is a standard that assigns a unique code to every character in any language. It has several parts: Core specification gives the general principles, Code charts show representative glyphs for all the Unicode characters, Annexes supply detailed normative information, and Character Database normative and informative data for implementers.

The basic concepts defined by the standard are:

- **Character**: the abstract concept e.g. `LATIN SMALL LETTER I`
- **Glyph**: the graphical representation of a character, e.g. i *i* **i** i *i*
- **Font**: a collection of glyphs
- **Codepoint**: the numeric representation of a character
  - represented with `U+` followed by the hexadecimal code e.g. `U+0069` for `'i'`
  - included in the range `U+0000` to `U+10FFFF` (21 bits)

The encoding is the mapping from code point to a byte sequence, decoding is the inverse. Unicode standard defines several alternative encoding (and decoding) options:

- **UTF-32** uses fixed width, 32 bits per code point, since it uses at most 23 bit and often just 8, it has a large memory overhead
- **UTF-16** is a variable width encoding, it represents:
  - codepoints from `U+0` to `U+d7ff` on 16 bits (2 bytes) and
  - codepoints from `U+10000` to `U+10ffff` on 32 bits (4 bytes)
- **UTF-8** is a variable width encoding, it represents:
  - codepoints `U+00` to `U+7f` are mapped directly to single bytes, i.e. ASCII transparent,
  - for the remaining code points, the high bit (0x80) marks multi byte character. Most non-ideographic codepoints are represented on 1 or 2 bytes e.g. `U+00C8` representing character 'è' is mapped to two bytes: `0xC3 0xA8`.

6.1.2. Class Charset

Class `Charset` allows handling different charsets and encodings in addition to the default Unicode, they are used for reading and writing.

It proides a few static methods to manage the available charsets in the system:

- `defaultCharset()`: returns the object corrsponding to the default charset
- `forName(..)`: returns the corresponding charset
- `availableCharsets()`: returns a map of all charsets by name

The predefined charsets available in any Java installation are:

- `US-ASCII`: 7-bit ASCII, a.k.a. `ISO646-US`, `ECMA-6`("7-bit coded character set" 1991)
- `ISO-8859-1`
  8-bit single byte ISO Latin No. 1, a.k.a. `ISO-LATIN-1`("8-bit single-byte coded graphic character sets - Latin alphabets No. 1 to No. 4" 1986)
- `UTF-8`
  8-bit multi byte UCS Transformation Format
- `UTF-16BE`
  16-bit UCS Transformation Fmt., big-endian
- `UTF-16LE`
  16-bit UCS Transformation Fmt., little-endian
- `UTF-16`
  16-bit UCS Transformation Fmt., w/byte-order mark

The `Charset` class provides two main methods:

- `ByteBuffer encode(CharBuffer)`: encodes a sequence of chars into a sequence of bytes
- `CharBuffer decode(ByteBuffer)`: decode a sequence of bytes into a sequence of chars

The encoding and decoding is performed through encode and decoder objects that can be created through factory methods:

- `getDecoder()`
- `getEncoder()`

The decoder and encoder objects have an internal state, e.g. awaiting next byte of a multi-byte representation.

Using a decoding scheme to decode a string encoded with a different scheme may lead to an **encoding mismatch**. For instance, character 'è' has Unicode codepoint `U+00C8` which is mapped in UTF-8 to two bytes: `0xC3 0xA8`, while IS0-8859-1 decoding interprets the above sequence as two distinct characters 'Ã¨'. Viceversa, 'è' in IS0-8859-1 is represented as 0xE8 which is an invalid character in UTF-8 (usually represented as)

6.2. Strings

There is no primitive string representation, there are three classes that represent strings:

- Class `String`, immutable, not modifiable version
- Classes `StringBuffer` and `StringBuilder`, mutable, modifiable versions

```
String s = new String("literal");
StringBuilder sb = new StringBuffer("literal");
```

6.2.1. Class String

Java redefines the operator `+` for strings. It is used to concatenate 2 strings, e.g. `"This is " + "a concatenation"`. It is important to remember that strings are immutable, therefore the application of the operator `+` creates a new string object with the result of the concatenation.

Operator `+` works also with other types, everything is automatically converted to a string representation using the `toString()` methods fo objects or the default representation of primitive types:

```
System.out.println("pi = " + 3.14);
System.out.println("x = " + x);
```

The two main string methods are:

- `int length()`: returns string length
- `boolean equals(String s)`: compares the contents of two strings
- `String toUpperCase()` Converts string to upper case
- `String toLowerCase()` Converts string to lower case
- `String concat(String str)` Creates a concatenation with the given string
- `int compareTo(String str)` Compare to another string returning:
  - < 0 : if this string precedes the other
  - $== 0$: if this string equals the other
  - > 0 : if this string follows the other
- `String subString(int startIndex)` `"Human".subString(2)` -> `"man"`
- `String subString(int start,int end)` Char start included, end excluded `"Greatest".subString(0,5)` -> `"Great"`
- `int indexOf(String str)` Returns the index of the first occurrence of str
- `int lastIndexOf(String str)` The same as before but search starts from the end

Example:

```
String h = "Hello";
String w = "World";
String hw = "Hello World";
String h_w = h + " " + w;
hw.equals(h_w)  // -> true
hw == h_w       // -> false
```

In addition String provides the static method:

- `String valueOf(..)`: converts any primitive type into a `String` Overloads defined for all primitive types.

6.2.2. Formatting

It is possible to use a format syntax similar to C `printf()` using two alternatives:

- `static String format(String fmt, ...)` is a static methods that builds a string using the format string,
- `String formatted(…)` builds format in the string is is called

Example formatting:

```
answer = String.format("%d",42);
answer = "%d".formatted(42);`
```

Format essentials:

`%[arg_index$][flags][width][.prec]conversion`

- arguments are positional unless `arg_index` is provided, it starts at 1
- `flags` can be:
  - `-`: left justified
  - `+`: include sign
  - `0`: 0 padding
  - `(`: Negative in parenthesis
- `width` indicates the min width
- `prec` defines the max width or decimal digits for floats
- `conversion` can be:
  - `b` boolean
  - `s` string
  - `d` integer
  - `f` decimal
  - `e` scientific

6.2.3. StringBuilder and StringBuffer

The classe `StringBuilder` and `StringBuffer` are method-level compatible classes. They represent a string of characters that is mutable and allows operation that modify the content. Can be converted to the corresponding `String` using the method `toString()`. The difference is that `StringBuilder` is non thread safe and non reentrant, this makes it more efficient, i.e. ~30% faster.

The main methods are:

- `append(String str)`: inserts str at the end of string
- `insert(int offset, String str)` Inserts str starting from offset position
- `delete(int start, int end)` Deletes character from start to end (excluded)
- `reverse()` Reverses the sequence of characters

They all return a `StringBuffer`/`StringBuilder` enabling method chaining.

6.2.4. Performance

The three alternative representations of strings exhibit very different performance behaviors.

Let us consider a very simple use case where many concatenations has to be perfomed to build the resulting final string.

```
String s="";
for(i=0;i<N;++i){
    s += i;
}
```

```
StringBuffer sb = new StringBuffer();
for(i=0;i<N;++i){
  sb.append(i);
}
```

```
StringBuilder sb = new StringBuilder();
for(i=0;i<N;++i){
  sb.append(i);
}
```

The three above code fragmets executed with `N=100000` yield the follwing elasped times:

Version	Elasped time	Memory used
String	1.3 s	500.0 MB
StringBuffer	2.9 ms	1.2 MB
StringBuilder	2.2 ms	0.8 MB

In addition to time performance it is important to remember that `+` instantiates a new object on each concatenation, thus also memory performance is significantly worse.

The huge advantage in using `String` is that it leads to much simpler code, faster to write and easier to understand.

> **Note**
>
> As a general programming advice: start writing your string manipulation using `String` and operator `+` this will make code faster to write and easier to understand. Later, if the code has relevant performance issues refactor it to use `StringBuilder` or `StringBuffer`, which are method-compatible. Use the latter only if thread safety is required.

6.2.5. String Pooling

Class `String` maintains a private static pool of distinct strings. The pool is managed through the method `intern()` which, when called:

- checks if any string in the pool `equals()` the argument
- if it finds one that string is returned
- otherwise it adds the string to the pool and returns it

For each string literal the compiler generates code using `intern()` to keep a single copy of the string with that specific content. This process is called *string internalization*. In practice, the code:

```
String ss1 = "Hello!";
```

Generates the same code as:

```
String ss1 = (new String(new char[]{'H', 'e', 'l', 'l', 'o', '!'}) ).intern();
```

On the first occurrence of a literal compiler creates the string and adds it to the pool. Upon later occurrences of the same literal, the compiler creates a string and through `intern()` returns a reference to the same single one in the pool.

6.3. Wrap-up

- Java characters are stored internally using a 16 bits unicode encoding
- Conversion to/from streams of bytes is managed by `Charset` objects
- `String` is immutable representation of strings
- `StringBuilder` and `StringBuffer` are mutable alternatives, significantly more efficient for string manipulation

7. Inheritance

A class can be declared a sub-type of another (*base*) class, the new (*derived*) class, implicitly contains (*inherits*), all the members of the class it inherits from.

The derived class can *augment* the base class structure with any additional member that it defines explicitly. Moreover, it Can *override* the definition of existing methods by providing its own implementation

The code of the new derived class consists only of the augmentations and overrides w.r.t. the base class.

Example base class

```
class Employee{
    String name;
    double wage;
    void incrementWage(){
    wage += 10;
  }
  void print(){
    System.out.println(name);
  }
}
```

In Java the inheritance relation between two classes is defined in the declaration of the derived class, using the keyword **`extends`** followed by the name of the base class.

An example of class `Manager` that inherits from `Employee` to augment it:

```
class Manager extends Employee{
    String managedUnit;                                                    ①
    void changeUnit(String u){                                             ②
    managerUnit = u;
  }

  public void print(){                                                     ③
      System.out.println (name + ", manages " + managedUnit);
  }

}
```

① a new attribute is added to those inherited from `Employee`
② a new method is added to those inherited from `Employee`

③ method `print()` overrides the one defined in class `Employee`

With the above defintion, it is possible in any method to write:

```
Manager m = new Manager();                                    ①

m.incrementWage();                                            ②
m.changeUnit("Section 21");                                   ③

m.print();                                                    ④
```

① a new Manager object is created
② even if the `Manager` class does not explicitly define an `incrementWage()` method it *inherits* it from `Employee`
③ method `changeUnit()` is not present in the base class but it has been added
④ the `print()` method is invoked, the version in `Manager` is used instead of the overridden one in `Employee`

7.1. Why inheritance

There are several motivation for using inheritance in OO languages.

The first and most obvious advantage of inheritance is **reuse**.

Often, a class consists of minor modifications of an already existing class, inheritance avoids code repetitions. As a consequence it enables *localization of code*, i.e. having the code specific to a given feature collected together. The results are:

- fixing a bug in the base class automatically fixes it in all the subclasses;
- adding a new functionality in the base class automatically adds it in the subclasses too;
- less chances of different (and inconsistent) implementations of the same operation.

A second important aspect, deriving from the principle of substitution, is **flexibility.**

Often, a program needs to treat several objects belonging to different classes in the same way. Two properties of inheritance in OO languages make this easy and seamless:

- *polymorphism* allows feeding algorithms with objects of different classes, provided they share a common base class;
- *dynamic binding* allows accommodating different behavior behind the same interface.

7.1.1. Generalization

Inheritance implies a relation of generalization or specialization. Generalization when moving from derived classes to base class, specialization when moving in the opposite direction. Let us consider, for instance, the example UML diagram shown in fig

Figure 7.1.: UML Class Diagram of Employee, Manager, CEO

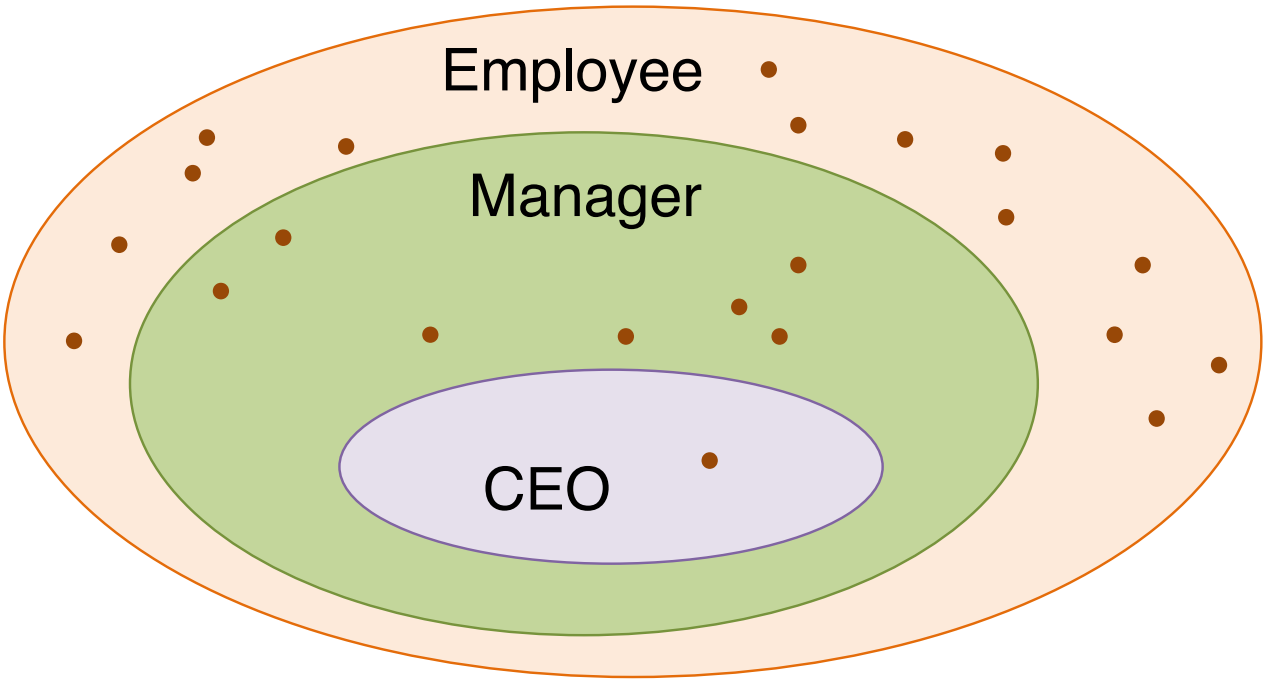


Figure 7.2.: Inheritance and subsets

The three classes, `Employee`, `Manager`, and `CEO` are linked by inheritance relationships. In particular `Manager` extends/inherits from `Employee`, and `CEO` extends/inherits from `Manager`. As a consequence the sets of objects instances of the three classes are related by subset relations as shown in Figure 7.2.

$$\mathsf{CEO} \subset \mathsf{Manager} \subset \mathsf{Employee}$$

In practice every CEO is also a manager and every manager is also an employee.

7.1.2. Inheritance graphs

Inheritance can exist on multiple leveles, as in Figure 7.1. Considering the classes as nodes and the *extends* relations as edges, we obtain a Directed Acyclic Graph (DAG).

In praticular when we consider a base class and all the classes that – either directly or indirectly – inherit from that class, the result is a tree graph that is usually called *inheritance tree*.

As a principle of good design it is important to remember that too deep inheritance trees might reduce the code understandability. In fact, to figure out the structure and behavior of a class in the tree, you need to look into each and every ancestor class

A commonly used measure to asses the complexity of an inheritance tree si the *Depth of Inheritance Tree* (*DIT*).

DIT is defined as the number of levels that exist below the root base class.

A general rule of thumb is to keep $DIT \leq 5$. Please note that this is an empirical limit that is not universally valid. It is useful, whenever the *DIT* in your code is above the threshold, to check the understandability of the code.

7.1.3. Terminology

The usual terminology adopted with respect to inheritance is:

- Class one above: parent class
- Class one below: child class
- Class one or more above: Superclass, Ancestor class, Base class
- Class one or more below: Subclass, Descendent class, Derived class

7.2. Polymorphism and Dynamic binding

7.2.1. Polymorphism

A reference of type T can point to – be dereferenced to – an object of type S if-and-only-if ether

- S is equal to T or
- S is a subclass of T.

More formally:

$$\text{type}(r) = T \wedge \text{deref}(r)\ \texttt{instance}_\circ\texttt{f}\ S \iff S = T \vee T\ \texttt{extends}\ S$$

where:

- r is a reference
- T and S are class types
- type is a function that links a reference to its type
- deref links a reference to the object it points to
- $\texttt{instance}_\circ\texttt{f}$ is the relationship between an object and its class
- `extends` is the relationship of inheritance between two classes

For instance, taking the classes defined above:

```
* Employee e;
* e = new Employee();                                                    ①
* e = new Manager();                                                     ②
```

① usual way, reference type and object type are the same (`Employee` = `Employee`)
② `e` is polymorphic and refers to an object of a derived class (`Manager extends Employee`)

The compiler treats indifferently objects of different classes, provided they derive from a common base class. Polymorphism can be leveraged, e.g., to collect objects that are instances of different classes into a single container:

```
Employee[] team = {
  new Manager("Mary Black",25000,"IT"),
  new Employee("John Smith",12000),
  new Employee("Jane Doe",12000),
  new CEO("Elma Moss",300000)
};
```

Polymorphism in a strongly typed language – such as Java – allows the compiler to perform strict static type checking.

The compiler performs a check on method invocation on the basis of the reference type, which is the only information available at compile time.

```
for(Employee it : team){
  it.print();
}
```

The compiler checks whether type of `it` (i.e.`Employee`) provides the method `print()` with no arguments. If that is true the call is considered correct. In fact `it` can point to any subclass of `Employee` and all them will inherit that method.

7.2.2. Dynamic Binding

When talkin about OO in general terms we say that the the methods of a class define the messages that can be received by the objects of that class. Up to now – before inheritance – we mixed the concepts of message and method.

When we consider the possibility for a derived class to *override* a method defined in a base class, the association between the message (i.e. method invocation) and method (i.e. method actual execution) is not unique anymore.

The association between the message (invoked method) and method (execution) is performed by the JVM at run-time using the *dynamic binding* procedure.

Dynamic binding procedure:

1. The JVM retrieves the effective class of the target object
2. If that class defines or overrides the required method, that version of the method is executed
3. Otherwise the parent class is considered and step 2 is repeated

Note that the procedure is guaranteed to terminate since the compiler checks the reference type class (a base of the actual one) defines the method.

Considering the following example:

```
for(Employee it : team){
  it.print();
}
```

Based on the effective class of the object referenced by `it` in each iteration, a different version of the method `print()` shall be executed. If the effective class is

- `Employee`, the class defines the original version which is executed,
- `Manager`, the class defines an override of the method which is executed,
- `CEO`, the class does not define any matching method, thus the parent class `Manager` is considered, which defines an override that is executed.

The important consequence of the dynamic binding procedure is that several objects from different classes, sharing a common ancestor class can be treated uniformly. It is possible to write algorithms for the base class (using the relative methods) and apply them unmodified to any subclass.

7.2.3. Substitutability principle

In practice polymorphism and dynamic binding enable the *substitutability principle* (Liskov and Wing 1994).

We can formulate the principle as: if S is a subtype of T, then objects of type T may be substituted with objects of type S without affecting the properties valid for the original objects.

This principle is also known as the Liskov Substitution Principle (LSP) from Barbara Liskov who first proposed it.

7.2.4. Inheritance vs. Duck typing

Dynamic binding can be applied in strict relation to inheritance or unrelated to it. This latter approach is used in other languages, e.g. Python.

The languages that use dynamic binding independently from inheritances are said to use **Duck typing**.

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck

The compiler does not perform any type check for method invocation. The correctness of method invocation is checked at run-time only. Invocation is correct if the effective class of the target object provides the required method (directly or inherited).

In languages using *duck typing* dynamic binding can result into a run-time error. While languages using strict type checking guarantee dynamic binding procedure always succeed.

7.2.5. Override rules

A method override must use exactly the original (overridden) method signature.

Visibility cannot be restricted since it would affect the correctness of the dynamic binding procedure. Although the method override may widen visibility (e.g. from package to public).

When a slightly different method is defined in a derived class, the compiler (and JVM) will not considered it as an override and therefore it will not be considere by the dynamic binding procedure. As a result, minor mistakes in typing might jeopardize correct behavior at run-time.

The annotation`@Override` before a method informs the compiler that a method is intended as an override. Thus the compiler generates an error if it is not a correct override, i.e. if the signature does not match that of method in a superclass.

Example of improper override:

```
class MiddleManager extends Employee{
  public void Print(){ ... }                                    ①
}
```

① the method is not an override since is stars with a capital letter while the original method in class `Employee` starts with a lowercase letter.

The result is that at run-time this variation will not be executed, and the version in the base class will be used instead.

The recommended way of defining an override is:

```
class MiddleManager extends Employee{
    @Override                                                   ①
    public void Print(){…}                                      ②
}
```

① the `@Override` annotation informs the compiler that the following method is intedend to be an override
② the compilers produces an error telling the method `Print()` should override a method present in a super class.

7.3. Casting

Java is a strongly typed programming language, i.e., each variable has a type and a variable can host only values of that type.

```
float f;
f = 4.7;                                                    ①
f = false;                                                  ②
Car c;
c = new Car();                                              ③
c = "car";                                                  ④
```

① legal assignment floating point value to FP variable
② illegal assignment: boolean value to FP variable
③ legal assignment: Car reference to Car variable
④ illegal assignment: string reference to Car variable

Cast with primitive types correspond to a type conversion, either explicit or implicit

```
int i = 42;
float f = i;                                                ①
f = (float) i;                                              ②
```

① implicit cast: 2's complement to floating point
② explicit cast

7.3.1. Upcast

Assignment from a more specific type (subtype) to a more general type (supertype) is called an **up-cast**. The term derives from the convetion in class diagram (see Figure 7.1) of representing base classes at the top and derived classes at the bottom, therefore a conversion form a derived to a base class moves *up*.

```
Employee e = new Employee(…);
Manager m = new Manager(…);
Employee em = (Employee) m;
```

Given the set interpretation of inheritance we have that:

$$\forall m \in \text{Manager} : m \in \text{Employee}$$

Therefore upcasts are always type-safe and are performed implicitly by the compiler. The cast `(Employee)` from the previous code be safely omitte:

```
Employee em = m;
```

It is importat to remember that reference type and object type are distinct concepts. When casts involve class types, we do not have any "conversion" of the value but just a change in reference type. In practice only the reference is affected while the pointed object remains totally unaltered. In the previous example the object referenced to by `em` continues to be of `Manager` type. Notably, in contrast, a primitive type cast involves a value conversion.

7.3.2. Downcast

Assignment from a more general type (supertype) to a more specific one (subtype) is called a **down-cast**.

```
Employee em = new ...() ;
Manager mm = (Manager)em  ;
```

$$\exists em \in \text{Employee} : em \notin \text{Manager}$$

Since it is possible that te cast is not correct, no automatic conversion is provided by the compiler. Down-casts must be explicit, so that they force the programmer to take responsibility of checking the cast is valid.

To access a member defined in a class you need a reference of that class type – or any subclass –. Therefore when the starting point is a generic (base class) reference, a down-cast can be required to access specific (derived class) members:

```
Employee emp = team[0];                    ①
String d = emp.getDepartment();            ②
Manager mgr = (Manager)team[0];            ③
d = mgr.getDepartment();                   ④
```

① simple assignment without casting
② syntaxError: The method `getDepartment()` is undefined for the type `Employee`
③ down-cast to `Manager`, correctness guaranteed by the developer
④ correct

Warning: the compiler trusts any downcast written by the programmer. The JVM checks type consistency for all reference assignments, at run-time, it ensures that the class of the object must be equal to the class of the reference or to any of its subclasses.

```
mgr = (Manager)team[1];                    ①
```

① `ClassCastException`: `Employee` cannot be cast to `Manager`

To ensure downcast safety is is possible to use the the **`instanceof`** operator: `aReference instanceof aClass` evaluates to `true` if the object pointed to by the reference can be cast to the class, i.e. if the object belongs to the given class or to any of its subclasses.

Example `instanceof`

↓ `instanceof` →	Employee	Manager	CEO
anEmployee	true	false	false
aManager	true	true	false
aCEO	true	true	true

A safe way of rewriting the above code is:

```
if(team[1] instanceof Manager){
  mgr = (Manager)team[1];
}
```

The above code idiom is not easy to read and redundant, for this reason since Java 16 the Java language allows the pattern matching instanceof. The new syntax allows writing the check with `instanceof` followed by a safe downcast can be written in a more compact and efficient form as:

```
if(team[1]instanceofManager mgr){
    // mgr is in scope here
}
```

7.4. Inheritance and Visibility (scope)

The classes related by inheritance follow the usual rules concerning visibility an access. Therefore a derived class is not allowd to access a base class `private` members.

```
class Employee {
  private String name;
  private double wage;
}

class Manager extends Employee {
  void print() {
    System.out.println  (  "  Manager  "  + name +   "     "   + wage);          ①
  }
}
```

① `name` and `wage` are not visible

Since it is quite common, for a derived class to require access to the base class members, specific visibility rules must be defined for such cases. For this reason an additional visibility level, marked by the modifier `protected`, is available.

The complete rules for visibily are:

- `public` members are always accessible,
- `protect` members dare accessible from classes in the same package and subclasses,
- *package* members are accessible from classes in the same package,
- `private` members are accessible only from within the declaring class.

In summary:

	Method in the same class	Method of other class in the same package	Method of subclass	Method of class in other package
private	yes			
package	yes	yes		
protected	yes	yes	yes	
public	yes	yes	yes	yes

7.5. Inheritance and constructors

The construction of an instance of a derived class must take into consideration the inherited parts. Since each object "contains" an instance of the parent class – i.e., the inherited attributes–, this part must be initialized first so that the derived class-specific parts can give that part for granted.

The Java compiler automatically inserts a call to the default constructor – i.e. any constructor without arguments – of the parent class. The call is inserted as the first statement of each child class constructor. Execution of constructors proceeds top-down in the inheritance hierarchy. As a consequence, when a method of the child class is executed (constructor included), the super-class is completely initialized already

As an example, given the follwing constructions that print a message:

```
class Employee {

  Employee() {
    System.out.println("ctor Employee");
  }
}
```

```
class Manager extends Employee  {
  Manager() {
    System.out.println("ctor Manager");
  }
}
```

```
class CEO extends Manager {
  CEO() {
    System.out.println("ctor CEO");
  }
}
```

The creation of a `CEO` object

```
CEO ceo = new CEO();
```

will produce the following outpu

```
ctorEmployee
ctorManager
ctorCEO
```

7.5.1. `super` (constructor)

The procedure described above works it there is a default constructor, defined explicitly or implicitly (provided by the compiler is no constructor is defined).

If a constructor with arguments is defined, the default constructor provided by the compiler "disappears", as a consequence, the derived class constructor implicitly invokes the base class constructor which cannot be resolved

```
class Employee{
  private String name;
  private double wage;

  Employee(String name, double wage){                                    ①
    name = n;
    wage = w;
  }
}
```

① an explicit constructor is defined, thus no default (void) constructor is not defined

```
class Manager extends Employee{                                          ①
}
```

① Error: no matching constructor in class Employee, the default constuctor is not available

The child class constructor must call the right constructor of the parent class, explicitly. It possible to use `super()` to invoke the right constructor of parent class and pass the appropriate arguments. It must be the first statement in a child class's constructor

Example:

```
class Manager extends Employee{
  private String unit;

  Manager(String n, double w, String u) {
    super(n,w);                                                    ①
    unit = u;                                                      ②
  }
}
```

① the constructor of the super class is invoked with the two required arguments `n` and `w`
② the inizialization of the specific attribute `unit`

When no `super` invocation is present in the constructor, the compiler automatically adds a `super()` that invokes the default constructor.

7.5.2. `super` (reference) example

When a method in a derived class overrides one in the base class, the latter is masked, i.e., the overridden method is invisible from the derived class as well as the users of the class. This might represent a problem if we wish to re-use the original overridden method from within the subclass.

To overcome this limitation, it is possible to use the `super` special reference that allow referring to members as defined in the super class.

```
class Employee{
  String name;
  public void print(){
    System.out.println  (name);
  }
}
```

```
class Manager extends Employee{
  private String unit;

  public void print(){
    super.print();
    System.out.println("\tmanages " + managedUnit);
  }
}
```

There are two special references always available in the methods of a class:

- `this` references the current object
- `super` references the current object as if it were a parent class instance

In conformace with the visibility rules, it is possible to redefine an attribute defined in the base class. The attribute defined in the derived class masks the one defined in the base class. To access the masked out attribute from the derived class, it is possible to use the `super` reference.

```
class Parent{
  protected int attr = 7;
}
```

```
class Child extends Parent {
  protected String attr = "hello";                                    ①
  void print(){
    System.out.println( super.attr );                                 ②
    System.out.println( attr );
  }
}
```

① the `attr` attribute mask the attribute with the same name in class `Parent`
② the `super` reference allows accessin the masked attribute

As far as good design practices, while method override is extremely useful and widely adopted, attribute *override* is not a recommended practice as it can create confusion.

7.5.3. `final` inheritance

The keyword `final` applied to a method makes it not overridable by subclasses. It can be useful when methods must keep a predefined behavior, e.g. method provide basic service to other classes.

The keyword `final` applied to a class makes it not extensible. It can be useful the class behavior must be preserved because it is essential to many other classes. For instance, `String` as well all wrapper classes are defined as `static` because defining a sub-class (whose behavior may substantilly differ from the base class) and passing the relative instances instead of, e.g. a string, might afffect the behavior of other classes.

7.6. Class Object

in Java there is a class called "`Object`", it look like an antinomy but is is not.

All classes are subtypes of `java.lang.Object`. Whenever a new class is defined without a base class – i.e. without the `extends` clause – the compiler automatically adds `extends Object`.

7.6.1. Why class `Object`

The first motivation to have a common base class to all classes defined in the language is **generality**. Any instance of any class can be seen as an `Object` instance, thus `Object` is the universal reference type, much like the `void*` in C.

Since each class is either directly or indirectly a subclass of `Object`, it is always possible to *upcast* any reference to `Object`.

As a consequence, we can collect heterogeneous objects into a single container using `Object` references

```
Object[] objects = {
          "First!",
          new Employee(),
          Integer.valueOf(42),
          22.0 / 7.0 // autoboxed to Double
};
for(Object obj: objects){
  System.out.println( obj );
}
```

Since `Object` references can point to objects but not to primitive types, wrappers must be used instead of primitive types.

A second motivation is to have a minimal univesal **common behavior**. Class `Object` defines some common operations which are inherited by all classes. Often, they are overridden in sub-classes.

The `Object` class defines the following methods:

- `toString()`, returns string representation of the object
- `equals()`, checks if two objects have same contents
- `hashCode()`, returns a unique code
- `clone()`, (`protected`) creates a copy of the current object
- `finalize()`, (`protected`) invoked by garbage collector upon memory reclamation.

7.6.2. `Object.toString()`

The method `toString()` returns a string representing the object contents. The default (`Object`) implementation returns "`ClassName@hashcode`", e.g. `org.Employee@af9e22`.

The typical implementation of the method returns the contents at – some – attributes. To avoid possible mistakes, it is recommended to us the `@Override` notation.

```
class Employee {
  // ...
  @Override
  public StringtoString(){
    return "Employee: " + name;
  }
}
```

More in general the `toString()` method is used whenever a string representation is required, e.g. when

- an object is passed to a print method
- an object is concatenated with a string

Listing 7.1 Implementation of the valueOf(Object) method in String

```
public static String valueOf(Object obj) {
  return (obj == null) ? "null" : obj.toString();
}
```

When invoking the `System.out.print( Object )` method, it implicitly calls the `toString()` the argument to obtain the string representation to be printed. In fact, all the print or write methods call the `valueOf()` method of class `String` that is reported below. This approach is more robust than directly calling `toString()` since it avoids a `NullPointerException` that would occur if the object to be printed is `null` and returns simply `"null"` in such a case.

The same approach is taken when the string concatenation operator `+` is applied to an object: the same `valueOf()` method is called, providing a robust way of concatenating object references that might possibly be `null`.

> **Note**
>
> It is a good design practice to override the `toString()` method for small objects whose representation can fit a limited number of characters. The string representation of an object is extremely useful during debugging. It can be printed or logged to keep track of what happens during execution. Moreover, most debuggers call the `toString()` method to show a representation of the objects.

7.6.3. `Object.equals()`

The method `equals()` is used to test the *content equality* of two objects. It is called on the first object to be compared while the second is passed as argument to the method: `o1.equals(o2)`.

The default implementation (in class `Object`) compares references:

Listing 7.2 Implementation of the equals() method in Object

```
public boolean equals(Object other){
    return this == other;
}
```

When overridden, it must respect a contract, i.e. a set of properties:

- Reflexive: `x.equals(x) == true`
- Symmetric: `x.equals(y) == y.equals(x)`
- Transitive: for any reference `x`, `y` and `z`:
  `x.equals(y) && y.equals(z)` $\implies$ `x.equals(z)`
- Consistent: multiple calls to `x.equals(y)` must consistently return the same value, this means that no information used in the comparison should be changed (immutables);
- Robust: `x.equals(null) == false`

An example of `equals()` override

```
@Override
public boolean equals(Object o){
  if(o instanceof Employee other)){
    return this.name.equals(other.name);
  }
  return false;
}
```

The above implementation relays the check of equality among employees to a check of equality of their names, i.e. it calls the `equals()` method of class `String`. It satifies the *equals contract*, in details:

- it is reflexive, symmetric, and transitive since `String.equals()` is reflexive, symmetric, and transitive,
- it is consistent if the `name` attribute is immutable,
- it is robust since `null instanceof Employee == false`.

7.6.4. Object.hashCode()

The `hashCode()` method returns a (fairly) unique code for the object that can be used as index in hash tables or as a quick approximation for equality.

The default implementation (Object) converts the internal address of the object into an integer. It must be overridden to return codes that are function of the object state.

When overridden it must comply with the `hashCode()` contract

- Consistent: `hashCode()` must return the same value, if no information used in equals() is modified.
- Equal compliant:
  - `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`
  - If two objects are not equal, then calling `hashCode()` *may* return distinct values, producing distinct values for not equal objects may improve the performance of hash tables

`hashCode()` vs. `equals()`

Condition	Required	Not Required (but allowed)
x.equals(y) == true	x.hashCode()== y.hashCode()	
x.hashCode() == y.hashCode()		x.equals(y)== true
x.equals(y) == false		-
x.hashCode() != y.hashCode()	x.equals(y)== false	

7.6.5. Variable arguments - example

The `Object` class is used as the universal reference type when passing a variable list of arguments with the `...` syntax. In such cases the list of arguments is available in the method as an `Object[]`.

Example of variable arguments list:

```
static void printList(String pre, Object... args){
  System.out.print  (pre + " ");
  boolean first = true;
  for(Object o :args){
    if(!first){
      System.out.print(", ");
    }else{
      first = false;
    }
    System.out.print(o);
  }
  System.out.println  ();
}

public static void main(String[]   args  ) {
    printList("List:", "A", 'b', 123,   "  hi\!  "  );
}
```

7.6.6. Array Covariance

The usage of the `Object` reference as a generic reference type for elements in arrays is based on the assumption of *co-variance* of arrays.

Given that we have two classes – T and S – related by inheritance, T `extends` S, what is the relationship among the arrays having element of the two types? There are three possibilities:

$$T \texttt{ extends } S \implies \begin{cases} T[] \texttt{ extends } S[] & \text{covariant} \\ S[] \texttt{ extends } T[] & \text{contravariant} \\ - & \text{invariant} \end{cases}$$

The choice made in Java is to make arrays covariant, therefore e.g. `String[] extends Object[]`. Covariance is ideal when reading content of arrays but is may create problems when writing. For instance, the following code is correct according to the Java compiler:

```
String[] names = {"one","two","three"};
Object[] objects = names;                    ①
objects[0] = Integer.valueOf(1);             ②
String s = names[0];                         ③
```

① due to covariance of arrays this is considered an up-cast

② this is a regular up-cast, but produces an `ArrayStoreException` error
③ if the previous asignment were correct this would be an error

In practice whan assigning to an element of an array, the JVM checks the compatibility of the value with the type of the elements.

While the choice of making arrays covariant allows writing code that is unsafe at run-time when writing in the array, it allows a great flexibility since it permits defining general methods accepting `Object[]`.

7.7. Abstract classes

Often, a superclass is used to define common behavior for children classes. In this case some methods may have no obvious meaningful implementation in the superclass.

Abstract classes can leave the behavior partially unspecified, i.e. some method – abstract – can provide no body. Since they are incomplete, abstract classes cannot be instantiated.

A class can be declared abstract using the `abstract` modifier. The same modifier can be used in front of the methods that are left unimplemented.

```
public abstract class Shape {
  private int color ;
  public void setColor ( int color ){
    this.color  = color ;
  }

  // to be implemented in child classes
  public abstract void draw();
}
```

In general when a class represent an abstract concept and no implementation can be devised for a method, declaring it `abstract` is much better than writing a warning message, such as: `System.err.println("Sorry, don't know how to draw this shape");`

Then the classes extending the abstract class must provide an implementation to the abstract methods.

```
public class Circle extends Shape{
  public void draw(){
    // body goes here
  }
}
```

```
Shape a = new Shape();                                    ①
Shape a = new Circle();                                   ②
```

① since `Shape` is abstract is cannot be instantiated, this is an error
② class `Circle` is concrete, therefore it can be instantiated

The `abstract` modifier marks the method as non-complete / undefined The modifier must be applied to all incomplete method and to the class A class must be declared abstract if any of its methods is abstract

A class that extends an abstract class should implement (i.e. override) all the base class abstract methods If any abstract method is not implemented, then the class must be declared abstract itself.

7.7.1. Template Method Pattern

Example: Sorter

Without the use of `abstract`

```
public class Sorter {
  public void sort(Object v[]){
    for(int i=1; i<v.length; ++i){
      for(int j=0; j<v.length-i; ++j){
        if(compare(v[j],v[j\+1])>0){
          Object o=v[j];
          v[j]=v[j+1]; v[j+1]=o;
        }
      }
    }
  }

  protected int compare(Object a, Object b){
    System.err.println  ("Someone forgot about the compare() method!");
    return 0;   // why not 42?
  }
}
```

What else could we do here?

```
public abstract class Sorter {
  public void sort(Object v[]){
    for(int i=1; i<v.length; ++i){
      for(int j=0; j<v.length-i; ++j){
        if(compare(v[j],v[j\+1])>0){
          Object o=v[j];
          v[j]=v[j+1]; v[j+1]=o;
        }
      }
    }
  }

  protected abstract int compare(Object a, Object b);
}
```

Example: StringSorter

```
public class StringSorter extends Sorter{
    @Override
  int compare(Object a, Object b){
    String sa = (String)a;
    String sb = (String)b;
    return sa.compareTo(sb);
  }
}
```

Sample usage:

```
Sorter ssrt= new StringSorter();
String[] v={"g","t","h","n","j","k"};
ssrt.sort(v);
```

Template Method Example

- Context:
  - An algorithm/behavior has a stable core and several variation at given points
- Problem
  - You have to implement/maintain several almost identical pieces of code

See slide deck on design patterns

7.7.2. Composite Pattern

Abstract Expression Tree

(img/J03-JavaInheritance13.wmf)

Expression Tree example

```
Expression e =

  new Operation('\*',

                  new Value(6),

                  new Operation('\+',

                                new Value(4),

                                new Value(3)));
```

```
 System.out.println(e.formula() \+ " = "
                                    \+e.eval());
```

```
public abstract class Expression{
  public abstract double eval();
  public abstract String formula();
}
 public class Value extends Expression{
      private double value;
      public Value(double v){ value = v; }
_@Override_
  public double eval() { return value; }
_@Override_
  public String formula() {
    return   String.valueOf  (value);
  }   }
```

```
 public class Operation extends Expression{
   private char op;
      private Expression left, right;
   public Addition(char o, Expression l,
                           Expression r){
          op = o; left=l; right=r;
      }
  public double eval() {
   switch(op){
```

```
    case '\+': return   left.eval  ()\+  right.eval  ();
   … }
  }
  public String formula() {
        return "(" \+   left.formula  () \+ " " \+ op \+
           " " \+   right.formula  () \+ ")";
}    }
```

Composite Pattern

- Context:
  - You need to represent part-whole hierarchies of objects
- Problem
  - Clients need to access a unique interface
  - There are structural difference between composite objects and individual objects.

7.8. Interfaces

A Java interface is a Special type of class where:

- Methods are implicitly abstract (no body)
- Attributes are implicitly static and final
- Members are implicitly public

It is defined with the keyword `interface` instead of `class`. Cannot be instantiated like abstract classes, i.e. no `new`. Can be used as a type for references

Interface example

```
public interface Expression{
  double eval();
  String formula();
}
```

When a class `implements` (in place of `extends`) an interface, must override all the interface methods – unless the class is declared abstract –.

```
public class Value implements Expression {
  @Override
  void value(){ … }

  @Override
  void formula(){ … }
}
```

Warning

In object-oriented jargon the general term *interface* is used to indicate the set of publicly available methods, or a subset, when talking about many *interfaces*. A Java `interface`, is a distinct though related concept. When a class implements an interface the methods defined in the interface constitute an *interface* of the class.

7.8.1. Interfaces and inheritance:

An interface:

- cannot extend a class
- can extend many interfaces

A class:

- can extend a single class (exactly one, if not explicit it is `Object`)
  - Single inheritance
- implement multiple interfaces
  - Multiple inheritance

```
class Person extends Employee implements Cloneable, Comparable {…}
```

Inheritance Classes & Interfaces

Table 7.4.: Summary of inheritance between classes and interfaces.

	Class	Interface
Class	extends(exactly one)	implements(up to many)
Interface	X	extends(up to many)

7.8.2. Anonymous Classes

- Interfaces can be used to instantiate anonymous local inner classes.

An anonymous inner class is defined within a method code, uses a variation of the `new` operator and provides the implementation of the interfaces method:

```
Iface obj = new Iface(){
  public void method(){  …  }
};
```

7.8.3. Static methods in interfaces

Interfaces can host `static` methods with the following restriction:

- Cannot refer to instance methods (without a reference), like in regular classes;
- Cannot change `static` attributes since they are `final` by default in interfaces;
- Can be overridden, since Java 8

7.8.4. Default methods

There is a specific type of methods in interfaces that can hava an implementatin, such methods are called **default methods**. This kind of methods has been available since Java 8.

The default methods:

- Can refer to arguments and other methods
- Cannot refer to non-static attributes, since they are unknown to the interface
- Can be overridden in implementing classes as any regular method

The main motivation for default methods is to provide the capability of injecting a specific behavior inside interfaces that would otherwise be pure declarations. This is similar to regular methods in classes, but unlike classes, they leverage multiple inheritance.

The specific use case that led to the definition of default methods is the need to enhance classe that implement pre-existing interfaces – from previou versions of Java – by adding new functionality and still ensuring compatibility with existing code written for older versions of those interfaces.

In fact, default methods let multiple inheritance, which was banned entering the Java language from the main door of classes, re-enter through the window of interfaces.

Therefore default methods may lead to conflicts among methods.

Let us considera a fictious example with two interfaces:

```
interface If1{
    int m1(inti);
    default void m2(){
            // do something...
    }
}
```

```
interface If2 {
  int m3(inti);
    default void m2() {
            // do something...
    }
}
```

And a class the implements both interaces:

```
class Cls implements If1, If2 {
  @Override
  public int m3(inti) { /* .. */ }

  @Override
  public int m1(inti) { /* .. */ }
}
```

The class has a duplicate default methods named `m2` without arguments that are inherited from the interfaces `If1` and `If2`.

To resolve the conflit, the class must provide its own version of the two methdos that takes precedence over the default implementations found in the two base interfaces:

```
class Cls implements If1, If2 {
  @Override
  public int m3(inti) { /* .. */ }

  @Override
  public int m1(inti) { /* .. */ }

  @Override
  public void m2() { /* .. */ }
}
```

7.8.5. Static interface attributes

Attributes in interfaces can be exclusively `static` and are automatically considered `final`, i.e. they only can define constants.

Due to the multiple inheritance of interfaces it is possible to encounter a conflicts among static attributes defined in different interfaces.

```
interface If1 {
  static int CONST=1;
  int m1(int i);
}
```

```
interface If2 {
    static int CONST=2;
        int m2(int i);
}
```

```
class Cls implements If1, If2 {
  @Override
  public int m1(int i) { return -1; }

  @Override
  public int m2(int i) {
    return CONST ;                                                    (1)
  }
}
```

(1) the field `CONST` is ambiguous since it matches in both interfaces!

The solution is to disambiguate using the interface name:

```
class Cls implements If1, If2 {
  @Override
  public int m1(int i) { return -1; }

  @Override
  public int m2(int i) {
    return If2.CONST ;                                                (1)
  }
}
```

7.8.6. Functional interface

An interface containing only one regular method – i.e. excluding default and static methods – is called a **functional interface**.

In a functional interface the method's semantics is purely functional, that is the result of the method is based solely on the arguments.

Functional interfaces can be used to perform behavioral parameterization.

A set of predefined functional interfaces are declared in the package `java.util.function`. There are specific variation for different primitive types as well as the generics version (see Generics).

The functional interfaces that work with `Object` are:

- `Consumer`
  - `void accept(Object value)`
- `Supplier`
  - `Object get()`
- `Predicate`
  - `boolean test(Object value)`
- `Function`
  - `Object apply(Object value)`
- `BiFunction`
  - `Object apply(Object l, Object r)`

Note that the above versions of the methods in the interfaces are simplified versions. The actual ones use Generics

There are also primitive-specific functions, the `int` versions are:

- `IntFunction`
  - `Object apply(int value)`
- `IntConsumer`
  - `void accept(int value)`
- `IntPredicate`
  - `boolean test(int value)`
- `IntSupplier`
  - `int getAsInt()`
- `IntBinaryOperator`
  - `int applyAsInt(int left, int right)`

7.9. Interface Usage Patterns

Interces serve several different purpose in Java programs:

- Allows alternative implementations
  - Define a common "*interface*"
- Provide a common behavior
  - Define method(s) to be called by algorithms
- Enable behavioral parameterization

  - Encapsulate behavior in an object parameter
- Enable communication decoupling
  - Define a set of callback method(s)
- Allow class flagging

7.9.1. Alternative implementations

- Context
  - The same module can be implemented in different ways by distinct classes with varying:
    * Storage type or strategy
    * Processing
- Problem
  - The classes should be interchangeable
- Solution
  - An interface defines methods with a well-defined semantics and functional specification
  - Distinct classes can implement it

Example: Complex numbers

```
public interface Complex {
  double real();
  double im();
  double mod();
  double arg();
}
```

Can be implemented using, e.g., either cartesian or polar coordinates

```
class ComplexRect implements Complex   {
  private double   im  , re;
  public ComplexRect(double re, double im) {
    this.im = im; this.re = re; }

  @Override public doublereal() { return re; }

  @Override public doubleim() { return im ; }

  @Override public doublearg() { return Math.atan2(im, re); }

  @Override public doublemod() { return Math.sqrt  (re * re + im * im  ); }
}
```

```
class ComplexPolar implements Complex {
    private double mod, arg;
    public ComplexPolar  (double   mod  , double   arg  ) {
                  this.mod = mod; this.arg = arg; }

  @Override public doublereal() { return mod * cos(arg); }

  @Override public doubleim() { return mod * sin(arg); }

  @Override public doublemod() { return mod; }

  @Override public doublearg() { return arg; }
}
```

Sample usage

```
Complex c1 = new ComplexRect(4,3);

System.out.println  (c1 +
" ->   Module   " + c1.mod() +
"    argument  : " + c1.arg());

Complex c2 = newComplexPolar(5,0.6435);
System.out.println  (c2 +
" -> Real " + c1.real() +
"    Imaginary  : " + c1.im());
```

Default methods can be useful to implements operation that are independent of the specific implmentation, example:

```
public interface Complex {
  //...
  default boolean isReal(){
        return im()==0;
  }
}
```

Interfaces can become the façade for alternative implementations. If alternatives are known in advance, it is possible to define a few static methods that can serve as factory methods. In this case the concrete implementations of the interface can even remain hidden withing a package.

```
public interface Complex {
  // ...
  static Complex fromRect(double re, double   im  ){
      return new ComplexRect(  re,im  );
  }
```

```
  static Complex fromPolar(double mod, double arg){
      return new ComplexPolar( mod,arg );
  }
}
```

Sample usage:

```
Complex c1 = Complex.fromRect(4,3);

System.out.println  (c1 +
" ->   Module    " + c1.mod() +
"    argument  : " + c1.arg());

Complex c2 = Complex.fromPolar(5,0.6435);
System.out.println  (c2 +
" -> Real " + c1.real() +
"    Imaginary  : " + c1.im());
```

7.9.2. Common behavior

- Context
  - An algorithm requires its data to provide a predefined set of common operations
- Problem
  - The algorithm must work on diverse classes
- Solution
  - An interface defines the required methods
  - Classes implement the interface and provide methods that are used by the algorithm

7.9.2.1. Common behavior: sorting

An meaingful example of common behavior is represented by sorting algorithms. Class `java.utils.Arrays` provides the static method `sort()` that is able to sort the element if an array.

Sorting primitive types can be carried on by specialized version of the sorting algorithm:

```
int[] v = {7, 2, 5, 1, 8, 5};
Arrays.sort(v);
```

When sorting object arrays, the algorithm requires a means to compare two objects. This can be achived through an interface providing a method for that purpose.

This is goal of the standard interface `java.lang.Comparable`. It is defined (roughly) as:

```
public interface Comparable{
    int compareTo(Object obj);
}
```

: Simplified `Comparable` interface. {#lst-comparable-simplified}

The method `compareTo` returns:

- <0 if `this` precedes `obj`
- =0 if `this` equals `obj`
- >0 if `this` follows `obj`

Note: the above definition is a simplified version, the actual declaration uses generics.

Example of Comparable implementation

```
public class Student implements Comparable{
  int id;

  Student(int id){ this.id =id; }

  @Override
  public int compareTo(Object o){
    Student other = (Student)o;
    return this.id - other.id;
  }
}
```

7.9.2.2. Common behavior: iteration

A second example of common behavior found in the standard Java libraries is the iteration on container objects.

The solution is based on the Iterator Pattern.

- Context
  - A collection of objects must be iterated
- Problem
  - Multiple concurrent iterations are possible
  - The internal storage must not be exposed
- Solution
  - Provide an iterator object, attached to the collection, that can be advanced independently

It is more complex since includes two distinct interfaces:

The `java.lang.Iterable` interface

```
java{#lst-iterable-simplified .java lst-cap="Simplified version of Iterable."}public
interfaceIterator{ public interface Iterable{   Iterator iterator(); }
```

The method `iterator()` returns a object whose class must implement the `Iterator` interface.

The `java.util.Iterator` interface is defined as:

```
java{#lst-iterator-simplified .java lst-cap="Simplified version of Iterator."}public
interfaceIterator{   boolean hasNext();   Object next(); }
```

The semantics of the `Iterator` is as follows:

- Initially the iterator is positioned before the first element
- `hasNext()` checks whether a next element is present
- `next()` returns the next element and advances by one position

Note: the above definition is a simplified version, the actual declaration uses generics.

The combination of the two interfaces allow writing a loop on the elements of a container:

```
Iterator it = seq.iterator()
while( it.hasNext() ){
  Object element = it.next();
  System.out.println(element);
}
```

In fact, any class implementing `Iterable` can be the target of a *for-each* construct, which uses the methods in the `Iterator` interface interface to iterate over the elements. Thus the loop above can be rewritten into an equivalent using the for-each construc:

```
for(Object element : seq  ){
  System.out.println(element);
}
```

The following class implements `Iterable` to allow interating on a sequence of random values.

The above class can be used to iterate on the values with a for-each statement:

```
RandomSeq seq = new RandomSeq(10);
for(Object e : seq){
    double v = ((Double)e).doubleValue();
    System.out.println(v);
}
```

Listing 7.3 Iterable class RandomSeq

```
class RandomSeq implements Iterable {
  private double[] values;
  public RandomSeq(int n){
    values = new double[n];
    for(int i=0; i<n; ++i)
      values[i] = Math.random();
  }

  public Iterator iterator() {
    return new Iterator(){
      private int next=0;

      public boolean hasNext() {
              return next < values.length; }

      public Object next() {
              return Double.valueOf( values[next++]);}
    };
  }
}
```

7.9.3. Behavioral parameterization

- Context
  - A generic algorithm is fully defined but a few given core operations that vary often
- Problem
  - Multiple implementations with significant code repetitions
  - Complex conditional structures
- Solution
  - The operations are defined in interfaces
  - Objects implementing the interface are used to parameterize the algorithm

A behavioral parameters is an object that is intended to provide a behavior that can be plugged into an existing algorithm. Usually it has no state but only operations. Most often the behavioral parameter implements a functional interface.

7.9.3.1. Strategy Pattern

- Context
  - Many classes or algorithm has a stable core and several behavioural variations
    - The detailed operation performed may vary

- Problem
  - Several different implementations for the variations are needed
  - Usage of multiple conditional constructs would tangle up the code
- Solution
  - Embed each variation inside a strategy object passed as a parameter to the algorithm
  - The strategy object's class implements an interface providing the operations required by the algorithm

For instance, a behavioral parameter can implements the standard functional interface `java.util.function.Consumer`

Listing 7.4 Simplified version of Consumer.

```
public interface Consumer{
  void accept(Object o);
}
```

It can be used to accept and process objects from an an array.

```
static void forEach(Object[] v, Consumer c){
  for(Object o : v){
    c.accept  (o);
  }
}
```

```
String[] v = {"A", "B", "C", "D"};
Consumer printer = new Consumer(){
  @Override
  public void accept(Object o){
    System.out.println(o);
  }
};
forEach(v, printer);
```

7.9.3.2. Comparator

The limitation of the sorting solution that uses the common behavior approach with the interface `Comparable` is that the method defines one specific criterion for ordering.

The solution is to used an approach based on the Strategy pattern, implemented by an overload of the `sort()` method that leverages the `java.util.Comparator` interface.

The semantics of the `compare()` method is similar to that of the `compareTo()` method, it returns:

- a negative integer if `a` precedes `b`
- 0, if `a` equals `b`

Listing 7.5 Simplified version of Comparator.

```
public interface Comparator{
  int compare(Object a, Object b);
}
```

- a positive integer if `a` succeeds `b`

Note: the above definition is a simplified version, the actual declaration uses generics.

```
public class StudentCmp implements Comparator{
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return a.id - b.id  ;
  }
}
```

The comparator can be used in the following way

```
Student[] sv = {new Student(11),
                           new Student(3),
                           new Student(7)};

Arrays.sort(sv, new Comparator(){
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return a.id - b.id;
    }
});
```

7.9.4. Communication decoupling

When dealing with complex object-oriented designs, separating senders and receivers of messages is important to:

- Reduce code coupling
- Improve reusability
- Enforce layering and structure

7.9.4.1. Observer - Observable

Observer Pattern

- Context:

  - The change in one object may trigger operations in one or more other objects

- Problem
  - High coupling
  - Number and type of objects to be notified may not be known in advance

- Solution
  - Define a base Subject class that provides methods to
    - Manage observers registrations
    - Send notifications
  - Define a standard Observer interface with a method that receives the notifications

The pair of the `Observer` interface and `Observable` class allow a standardized interaction between an objects that needs to notify one or more other objects. They are both defined in package `java.util`.

- Class `Observable` manages:
  - registration of interested observers by means of method `addObserver()`
  - sending the notification of the status change to the observer(s) together with additional information concerning the status (event object).

- Interface `Observer` allows:
  - Receiving standardized notification of the observer change of state through method `update()` that accepts two arguments:
    - Observable object that originated the notification
    - Additional information (the event object)

Sending a notification from an observable element involves two steps: * record the fact the the status of the observable has changed, by means of method `setChanged()` , * send the actual notification and provide additional information (the event object), by means of method `notifyObservers()`

Warning: Observer-Observable have been deprecated in Java 9 because they exhibit a few limitations.

7.9.5. Flagging interface idiom

- Context:
  - A set of classes is treated similarly but a subset must be treated differently

- Problem:
  - Different objects must be identified at run-time
  - Adding a flag attribute would impact all classes

- Solution:
  - Let different classes implement an emptyflagginginterface
  - Check at run-time using instanceof

7.9.5.1. Interface `Cloneable`

In Java implementing `Cloneable` flags as safe making a field-for-field copy of instances. The `Object.clone()` method, if the class is flagged, makes a field-for-field copy of the object. Otherwise it throws a `CloneNotSupportedException` error.

By convention, classes that implement this interface should override `Object.clone()`. The override should get a copy using `super.clone()` and possibly modify fields on the clone object before returning it.

7.10. Lambda Functions and Methods References

The definition of anonymous inner classes is very common when using functional interfaces in Java idiomatic ways.

Let us conside, for instance the `Consumer` interface reported in Listing 7.4. The code required to instantiate an anonymous implementation is the following:

```
Consumer printer =                                                    ①
new Consumer(){                                                       ②
  public void accept(Object o){                                       ③
    System.out.println(o);                                            ④
  }
};
```

① a declaration of the `Consumer` reference
② the `new` statement with the same interface name as in the declaration
③ the functional method declaration (same as in the original interface)
④ the method body, i.e. actual implementation of the method.

The only fragments of code really useful are the declaration and the method body, all the rest is just boilerplate code. The boilerplate code has several drawbacks:

- requires time to be written,
- can contain mistakes,
- makes the code harder to read.

To reduce the boilerplate code to implement functional interfaces since Java 8, two new constructs have been introduce.

The **lambda expressions** that let the previous code be reduced to:

```
Consumer printer =                                                    ①
    o -> System.out.println(o);                                       ②
```

The **method references** that let the previous code be reduced to:

```
Consumer printer =                                                    ①
    System.out::println;                                              ②
```

7.10.1. Lambda expressions

A lambda expression is equivalent to the definition of anonymous inner class for functional interfaces. The Lambda expression syntax

parameters `->` *body*

Where:

- the parameters can be
  - None: `()`
  - One: e.g., `x` or `(x)`
  - Two or more: `(x, y)` must be enclosed in parentheses

  The parameter types can be omitted, in that case they are inferred from assignee reference type
- the body
  - Expression: e.g. `x + y`
  - Code Block: e.g. `{ return x + y; }`

The parameter types are usually omitted. Compiler can infer the correct type from the context, specifically the assignee reference type. They match the parameter types of the the functional interface only method

As an example, it is possible to write a comparator with a lambda expression:

```
Arrays.sort( sv,
  (a,b)-> ((Student)a).id - ((Student)b).id
);
```

Vs.

```
Arrays.sort(sv, new Comparator(){
  public int compare(Object a, Object b){
      return ((Student)a).id - ((Student)b).id;
}});
```

7.10.2. Method reference

A method reference is a compact representation of functional interface that relays the invocation to single method.

Method reference syntax

Container `::` *methodName*

There are several different types of method references based on the type of the container and the type of method

Kind	Example
Static method	`Class::staticMethodName`
Instance method of a given object	`object::instanceMethodName`
Instance method of an object of a given type	`Type::methodName`
Constructor	`Class::new`

7.10.2.1. Static method reference

Like a C function, the parameters of the functional method are mapped to the arguments of the static method.

For instance, using the standard `DoubleBinaryOperator` interface:

```
interface DoubleBinaryOperator{
  double applyAsDouble(double a, double b);
}
```

It is possible to define a functiona interface implementation that takes the maximum of two numbers:

```
DoubleBinaryOperator combine = Math::max;
double d=combine.applyAsDouble(1.0, 3.1);
```

The alternative implementation as a lambda expression is:

```
DoubleBinaryOperator combine = (a,b) -> Math.max(a,b)
double d=combine.applyAsDouble(1.0, 3.1);
```

7.10.2.2. Instance method of object reference

The functional method arguments are mapped to the arguments of the target method of the object.

For instance using the following functional interface:

```
interface IntToCharFunction{
  char apply(int value);
}
```

We can define a functional interface implementation that convers a decimal digit to an hexadecimal digit:

```
String hexDigits = "0123456789ABCDEF";
IntToCharFunction hex = hexDigits::charAt;
System.out.println  ("Hex for 10 : " +  hex.apply(10));
```

The alternative implementation as a lambda expression is:

```
String hexDigits = "0123456789ABCDEF";
IntToCharFunction hex = ch -> hexDigits.charAt(ch);
System.out.println  ("Hex for 10 : " +  hex.apply(10));
```

7.10.2.3. Instance method reference

The functional method arguments are mapped as follows:

- the first argument is mapped to the target object on which the method is invoked
- the possible remaining arguments are mapped to the target method arguments

For instance using the following functional interface:

```
interface StringToIntFunction {
    int apply(String s);
}
```

We can define a functional interface implementation that returns the length of a string:

```
StringToIntFunction len = String::length;
for(String s : words){
    System.out.println(len.apply(s));
}
```

The equivalent using lambda expression would be:

```
StringToIntFunction len = s -> s.length();
for(String s : words){
    System.out.println(len.apply(s));
}
```

7.10.2.4. Constructor reference

The functional method arguments are mapped to the arguments of the target class constructor. The return is a new object of the given type.

For instance using the following functional interface

```
interface IntegerBuilder {
        Integer build(int value);
}
```

It is possible to create a functional implementation that converts an int to an Integer

```
IntegerBuilder builder = Integer::new;
Integer i = builder.build(1);
```

The equivalent code using a lambda expression is

```
IntegerBuilder builder = i -> new Integer(i);
Integer i = builder.build(1);
```

7.10.3. Lambda Implementation

Lambda expression are implemented through method references. For each expression, a new method named `lambda$#` is created, the body contains the lambda code. The method is added to the declaring class. The compiler places a reference to that method in place of the lambda.

Lambda vs. Method reference

A method reference allows direct call.

For insance the followig code calls directly the `method()`

```
Runnable methRef = Class::method;
methRef.run();
```

and is exactly equivalente to a direct call:

```
method();
```

A lambda expression makes an indirect call, through the fictious method that is generated by the compiler:

```
Runnable lambda = ()->method();
lambda.run()
```

Calls the method `lambda$0()` that is defined as:

```
void lamda$0(){ method(); }
```

Which, in turn, calls the target `method()`. In this case if we look at the call stack we can observe:

- the caller method
- the lambda expression method
- the callee method

In general method references are preferrable because they are more efficient.

7.11. Practical Design Reflections

7.11.1. Inheritance vs. composition

- Reuse can be achieved via: *Inheritance
  - The reusing class inherits reused members that are available as own members
  - Clients can invoke directly inherited methods *Composition
  - The reusing class has the reused methods available in an included object (attribute)
  - Clients invoke new methods that delegate requests to the included object

(img/J03-JavaInheritance20.wmf)

(img/J03-JavaInheritance21.wmf)

Observer w/Inheritance

(img/J03-JavaInheritance22.wmf)

Observer w/Composition

(img/J03-JavaInheritance23.wmf)

Observer subject w/inheritance

```
public class Subject extends Observable{
  String  prop="ini";
  public void setProp(String val){
    setChanged();
    property =val;
    notifyObservers("theProp");
  }
}
```

Observer subject w/composition

```
public class Subject {

  PropertyChangeSupportpcs = newPropertyChangeSupport(this);
  String  prop="ini";

  public voidsetProp(String val) {
    String old = prop;
    property =val;
    pcs.firePropertyChange("theProp",old,val);
  }

  // delegation:

  public void addObs(PropertyChangeListener l){
    pcs.addPropertyChangeListener("theProp",l);
  }
}
```

Observer with inheritance

```
public class Concerned implementsObserver{
  @Override
  public void update(Observable src, Object arg) {
    System.out.println("Variation of " + arg);
    }
}
```

Observer with composition

```
public class Concerned implements PropertyChangeListener{
  @Override
  public voidpropertyChange( PropertyChangeEvent evt) {
    System.out.println("Variation of " + evt.getPropertyName());
  }
}
```

7.11.2. Algorithm variability

- Common behavior idiom
  - The variability is bound to the type of objects processed by the algorithm
  - Behavior is implicit in the data classes
  - Less flexibility
- Strategy pattern
  - The variability is implemented through behavioral objects (strategies)

  - Behavior is explicitly defined upon invocation
  - More flexibility

Sorting flexibility

```
class Student implements Comparable { //…
  public int compareTo(Object s){
    return id - ((Student)s).id;
  }
}
```

```
Arrays.sort(students); // <- implicit
```

Comparable provides an explicit strategy

```
// explicit strategy
Arrays.sort(students,(a,b)->{ // ascending
   return ((Student)a).id - ((Student)b).id;
});

Arrays.sort(students,(a,b)->{ // descending
   return ((Student)b).id - ((Student)a).id;
});
```

7.12. Wrap-up

- Inheritance
  - Objects defined as sub-types of already existing objects. They share the parent data/methods without having to re-implement
- Specialization
  - Child class augments parent (e.g. adds an attribute/method)
- Overriding
  - Child class redefines parent method
- Implementation/reification
  - Child class provides the actual behaviour of a parent method
- Polymorphism
  - The same message can produce different behavior depending on the actual type of the receiver objects (late binding of message/method)
- Interfaces provide a mechanism for
  - Constraining alternative implementations

  - Defining a common behavior
  - Behavioral parameterization

- Functional interfaces and lambda simplify the syntax for behavioral parameterization

8. Exceptions

The goal of exceptions is tor eport anomalies, by delegating error handling to higher levels. Methods detecting anomalies might not be able to recover from an error, often a caller method can handle errors more suitably than the detecting method itself.

An important advantage of exceptiosn is the ability to localize error handling code and to separate it from operating code. As a result the operating code is more readable and error handling code is collected in a single place, instead of being scattered.

8.1. Anomalies management

The general process for dealing with anomalies in programs is:

1. Detection: check conditions revealing an anomaly
2. Signaling: inform the caller about the anomaly
3. Dispatch: receive and redirect the anomaly signal
4. Handling: perform operation to address an anomaly

The main error signaling techniques are:

- Program abort (handling): abrupt termination of the execution
- Special value: return a special value to indicate error
- Global status: set a global variable containing error information
- Exceptions: throw an exception

8.1.1. Abort

If a non-remediable error happens, in Java it is possible to call `System.exit()`. It aborts program execution, the JVM does not perform any cleanup or resource release. A method causing an unconditional program interruption is not very dependable (nor usable). In addition it is not testable because the abort would terminate the JUnit tesing session.

> **Note**
>
> The reccomendation is to **never ever** call `System.exit()` in your programs

8.1.2. Special value

When an error happens a method can return a special value. Special values are distinct from normal values returned by the method.

Exampe of special values in standard libraries are:

- `String.indexOf()` that returns `-1` if the substring is not found, e.g. `"ABCD".indexOf("F")`
- the `Math` methods that can return `Double.NaN` if the operation is not possivle, e.g. `Math.pow(-1, 0.5)`

The main issues with this approach is that it is not always possible to identify a subset of values that can be used to signal an anomaly or error. For instance, when we try to print or concatenate a `null` string, the result is `"null"` which is difficult to distinguish from a string that effectively correspond to the four character `"null"`.

The error handling code in this case look like:

```
if( someMethod() == ERROR ) // acknowledge
   //handle the error
else
   //proceed normally
```

Developer must remember value/meaning of special values to check for errors.

In general the code can quickly get messy to write and hard to read when multiple errors must be handled.

In addition, only the direct caller can intercept errors and there is no simple delegation to any upward method unless further additional code is added.

```
int readFile() {
   int f = open();
   if (f == OPEN_ERROR)
      return -1;
   int n=size();
   if ( n == SIZE_ERROR)
      return -2;
   Object m = alloc();
   if (m = ALLOC_ERROR) {
      close the file;
      return -3;
   }
   String s = read();
   if (s == READ_ERROR) {
      close the file;
      return -4;
   }
   close();
```

```
   return 0;
}
```

8.1.3. Global error variable

In C many function set the global variable `errno` to signal that an error occurred during an operation, see: the man page for errno.

In Java, such error signaling approach is never used in standard libraries or in common applications.

8.2. Exception syntax

An example of code that uses exception instead of, e.g. special return values, looks like:

```
try {
   int f=open();
   int n = size();
   Object o = alloc(n);
   String s = read();
   close();
} catch (fileOpenFailed) {
        doSomething;
} catch (sizeDeterminationFailed) {
        doSomething;
} catch (memoryAllocationFailed) {
        doSomething;
} catch (readFailed) {
        doSomething;
} catch (fileCloseFailed) {
        doSomething;
}
```

The code is much cleaner, the operation code is all together and not dispersed with error handling code, which is collected separately. Overall it is easier to read and understand.

The code detecting the the error will throw an exception, it can be either developers' code or a library. At some point, up in the hierarchy of method invocations, a caller will intercept and handle the exception (`try-catch`). In between, dispatching methods can relay the exception (complete delegation) or intercept and re-throw (partial delegation)

Java provides three constructs to manage exceptions:

- `throw` operator throws an exception and starts the exception handling procedure
- `throws` declaration, declares that a method can possibly generate an exception
- `try-catch` statement introduces code to watch for exceptions and defines the exception handling code

It also defines a new type the `Throwable` class that is the base class for every exception. It is used rarely and the most commonly used class is `Exception` that extends it.

The anomali management with exceptions requires:

- identify or define an exception class that will represent the anomaly/error
- explicitly declare that some methods are potential sources of exception
- in such methods as usual check condition, and if an anomly is detected, create an exception object and throw it

Example of anomaly management with exception, concerning the `Stack` class. First of all, a new exception class `EmptyStack` can be defined:

```
public class EmptyStack extends Exception{}
```

Then the class Stack can use this class to signal an anomaly:

```
public class Stack{
    //...
    public int pop() throws EmptyStack {                    ①
        if(size == 0) {
            EmptyStack e = new EmptyStack();                ②
            throw e;                                        ③
        }
        return stack[--next];                               ④
    }
}
```

① Declaration `throws` states that the method can throw an `EmptyStack` exception
② Create an exception object, as a regular object
③ Operator `throw` throws the exception and terminates the method execution
④ in case of exception this statement is not executed

When an exception is thrown, the execution of the current method is interrupted immediately. The code immediately following the `throw` statement is not executed, it behaves like a `return` statement the difference is that the execution does not return to the caller but the catching phase starts.

If a method might generate an exception, it must must declare it in its signature with the declaration `throws`. All exception type(s) are listed after the throws keyword. It is mandatory to declare exceptions thrown both directly by the method, or by called methods and relayed. The declartion allow checking that the exception is somehow handled by caller.

The code calling a method that can throw an exception must handle it, typically using the `try-catch` statement:

```
static void main(String[] args){
    Stack s = new Stack();
    try{                                                    ①
        int i = s.pop();
```

```
    }catch(EmptyStack e){                                                    ②
        System.err.println("Empty stack: cannot pop");                       ③
    }
}
```

① `try` introduces that code where exception might be thrown
② `catch` intercept and exception and introduces the
③ code that handle the exception

Once an exception is thrown the normal execution is suspended. The thrown exception "*walks back*" the call stack until either:

- it is caught by one of the calling methods, or
- it overtakes `main()`, in this case the JVM prints the exception (and the full stack trace) and terminates execution.

8.3. Handling exceptions

When a fragment of code can possibly generate an exception, the exception must be dispatched:

- Relay the exception and let it propagate up the call stack, then the method must have a throws declaration,
- Catch, stop the exception, and handle it Code enclosed in `try{}catch(){}` statement
- Catch, partially handle, and re-throw

8.3.1. Relay

```
class Dummy {
  Stack st = new Stack();
  public int foo() throws EmptyStack{                                        ②
    int v = st.pop();                                                        ①
    return v + 1;
  }
}
```

① the method `pop()` can throw an exception
② the exception is relayed to the caller by method `foo()`

Exception not caught can be relayed until the `main()` method that relays it to the JVM that eventually catches is and terminate the program after printing the exception details.

8.3.2. Catch and handle

```
class Dummy {
  Stack st;
  public int foo(){
    try{                                                    ②
      int v = st.pop();                                     ①
      return v + 1;
    } catch (StackEmpty se) {                               ③
       // do something
    }
    return 0; // default value
} }
```

① the method `pop()` can throw an exception
② the `try` keyword introduces the code that can throw exceptions
③ the `catch` intercepts the exception and start handling it

8.3.3. Catch and re-throw

```
class Dummy {
  Stack st;
  public void foo() throws EmptyStack{                      ④
    try{
      int v = st.pop();                                     ①
      return v + 1;
    } catch (StackEmpty se) {                               ②
      // intermediate handling
      throw se;                                             ③
    }
} }
```

① the method `pop()` can throw an exception
② the `catch` intercepts the exception and start handling it
③ after partial processing the exeption is re-thrown
④ the method must declare the re-thrown exception

8.3.4. Exceptions and loops

For errors affecting a single iteration, the try-catch blocks is nested in the loop. In case of exception the execution goes to the catch block and then proceed with the next iteration.

```
while(true){
  try{
    // potential exceptions
  }catch(AnException e){
```

```
    // handle the anomaly
  } // and continue with next iteration
}
```

For serious errors compromising the whole loop, the loop is nested within the try block. In case of exception, the execution goes to the catch block, thus exiting the loop.

```
try{
    while(true){
        // potential exceptions
    }
}catch(AnException e){ // exit the loop and …
  // handle the anomaly
}
```

8.4. Exception classes

The base class for all exception classes is `Throwable`. It contains a snapshot of the call stack that is initialized by the constructor. May contain a message string that is intended tot provide information about the anomaly. May also contain a cause, i.e. a reference to another exception that caused this one to be thrown.

The main methods in `Throwable` are:

- `getMessage()` returns the error message associated with the exception
- `getCause()`: return a possible other exception that caused this one
- `printStackTrace()`: prints the stack trace until the place when the exception was created, this is the method called by the JVM on uncaught exceptions.

8.4.1. Checked and unchecked

Regular exceptions are called **checked exceptions**, they usually extend `Exception` and are checked by the compiler. When a `throw` statement or a methdo with `throws` declaration is found the compiler checks if that code is within a matching `try-catch` or the method `throws` those exceptions. If that is not the case the result is an **unhandled exception** error by the compiler.

There is special category of exceptions called **unchecked exceptions**, they do not need to be declared, i.e. theyr are not checked by the compiler. Typically **unchecked exceptions** generated by JVM when performing run-time checks, e.g. `NullPointerException`. Thus their generation is not foreseen (can happen everywhere), therefore to avoid exception ridden code they have been declared as unchecked.

Java consider **unchecked exceptions** all the exception classes that extend either `Error` or `RuntimeException`.

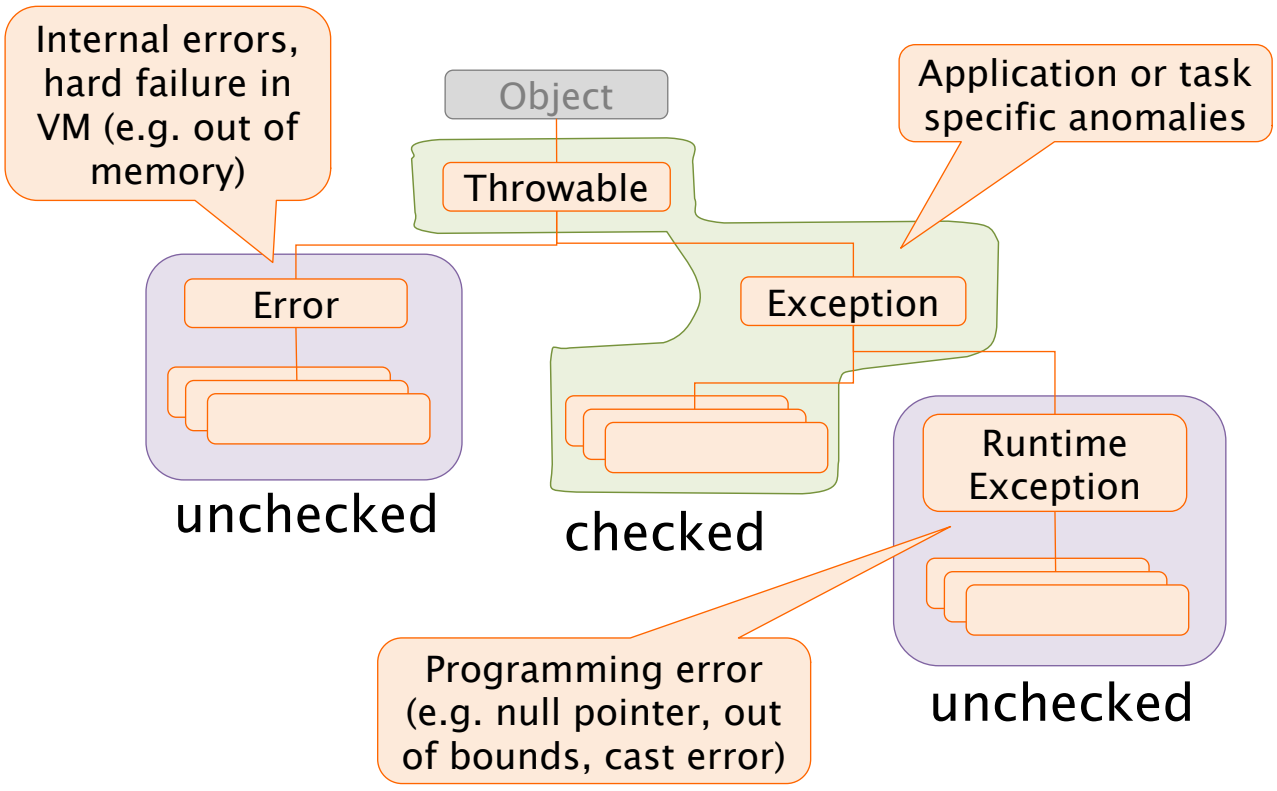


Figure 8.1.: Hierarchy of exception classes

8.4.2. Exceptions hierarchies

Exception classes examples

- `Error`
  - `OutOfMemoryError`
- `Exception`
  - `ClassNotFoundException`
  - `InstantiationException`
  - `IOException`
  - `InterruptedException`
- `RuntimeException`
  - `NullPointerException`
  - `ClassCastException`

Application specific exceptions represent anomalies specific for the application. Usually such exceptions extend `Exception`. They can be caught separately from the predefined ones, thus allowing finer-grained control than using just `Exception`.

In general it is important to used specific application specific exceptions that contain explanatory messages. Exceptions are like stones, when they hit you, they immediately matter because they exists and are thrown, then for their meaning and message.

8.4.3. Unchecked and loops

```
String[] strings = {"1","2","III","4","V","6"};
int sum = 0;
for(String s : strings) {
  sum += Integer.parseInt(s);                                    ①
}
System.out.println("Sum: " + sum);
```

① At runtime generates a `NumberFormatException: For input string: "III"`

```
String[] strings = {"1","2","III","4","V","6"};
try{
  int sum = 0;
  for(String s : strings) {
    sum += Integer.parseInt(s);
  }
  System.out.println("Sum: " + sum);
}catch(Exception e){
  System.err.println("Error!");
}
```

```
String[] strings = {"1","2","III","4","V","6"};
int sum = 0;
for(String s : strings) {
  try{
    sum += Integer.parseInt(s);
  }catch(NumberFormatException e){
    System.err.println("Wrong: "+s);
  }
}
System.out.println("Sum: " + sum);
```

8.5. Multiple exception

8.5.1. Nested `try`

Try/catch blocks can be nested, e.g. because error handlers may generate new exceptions.

```
try{
    /* Do something */
}catch(…){
      try{ /* Log on file */ }
      catch(…){ /* Ignore */ }
}
```

Unchecked and loop

```
sum = 0;
for(String s : strings) {
  try {
    sum += Integer.parseInt(s);
  }catch(NumberFormatException nfe) {
    try {
      sum += parseRoman(s);
    }catch(NumberFormatException re) {
      System.err.println("Wrong " + s);
} } }
System.out.println("Sum: " + sum);
```

8.5.2. Multiple `catch`

Capturing different types of exception is possible with different `catch` blocks attached to the same `try`

```
try {
 ...
}
catch(StackEmpty se) {
    // here stack errors are handled
}
catch(IOException ioe) {
  // here all other IO problems are handled
}
```

The matching rules state that only one handler is executed, the first one matching the thrown exception. A `catch` matches an exception if it is `instanceof` the catch's exception class.

As a consequence, catch blocks must be ordered by their "*generality*": from the most specific (derived classes) to the most general (base classes). Placing the more general first would obscure the more specific, making them unreachable and producing a compilation error.

For instance if `open` and `close` can generate a `FileError` and `read` can generate a `IOError`.

```
System.out.print("Begin");

File f = new File("foo.txt");
try{
  f.open();
  f.read();
  f.close();
}catch(FileError fe){
  System.out.print("File err");
}catch(IOError ioe){
  System.out.print("I/O err");
}

System.out.print("End");
```

If `close` fails, "`File error`" is printed and eventually program terminates with "End"

If `read` fails, "`I/O error`" is printed and eventually program terminates with "End"

8.6. `finally` clause

The keyword finally introduces a code block that is executed in any case

- No exception
- Caught exception
- Uncaught exception, both checked and unchecked

It does not work in case of `System.exit()` which instantly terminates the execution.

Some objects consume OS resources, such as input/output streams, db connections, etc. Such resources are limited, e.g., a program can open only a given number of files at once. Therefore those objects should be closed as soon as possible to free shared and limited resources.

A typical example is a file connection used for reading:

```
String readFile(String path) throws IOException{
  FileReader fr = new FileReader(path);

  int ch = fr.read();                                                    ①
  // …
  fr.close();
  return String.valueOf(ch);
}
```

① in case of exception in `read()` the method terminates without executing the `close()`

A possible alternative is

```
String readFile(String path) throws IOException {
  try {
    FileReader fr = new FileReader(path);
    int ch = fr.read();
    // …
    fr.close();
    return String.valueOf(ch);
  } catch(IOException e){
    fr.close();
    throw e;
  }
}
```

This code is more complex and does not close in case of other exceptions. Therefore the right solution is to use a `finally` clause:

```
String readFile(String path) throws IOException {
  try {
    FileReader fr = new FileReader(path);
    int ch = fr.read();
    // …
    fr.close();
    return String.valueOf(ch);
  }finally {
    if(fr!=null) fr.close();                                             ①
    throw e;
  }
}
```

① the file is close in any possible case

The **try-with-resource** construct has been defined (in Java 7) to make the latter syntax simpler. It is based on the `AutoCloaseable` interface that is defined as follow

```
public interface AutoCloseable{
  void close();
}
```

The `AutoCloaseable` interface serves the double function of common behavior and flagging to enable the automatic closing of resources.

The general syntax with *try-with-resource* construct is:

```
try( AutoCloseableType resource = new AutoCloseableType() ){
    // ...
}
```

The `AutoCloaseableType` is any class that implements the `Autocloseable` interface. The compiler automatically adds a `finally` block that closes the resource.

Therfore, the above code then can be re-written as:

```
String readFile(String path) throws IOException {
  try(FileReader fr = new FileReader(path)){
      int ch = fr.read();
      // ...
      fr.close();
      return String.valueOf(ch);
  }
}
```

Since `FileReader` implements `Autocloseable` the code is safe and the resource will be automatically released whatever it happens.

8.7. Defensive Programming

Defensive programming is a proactive strategy for making software more robust. It consists in anticipating possible issues and adding safeguards to prevent or clearly signal failures. In fact most of software bugs stem from edge cases that were not considered by tge developers. Defensive coding reduces surprises.

8.7.1. Fail Fast

> Some people recommend making your software robust by working around problems automatically.
>
> This results in the software "failing slowly." The program continues working right after an error but fails in strange ways later on.
>
> A system that fails fast does exactly the opposite: when a problem occurs, it fails immediately and visibly. Failing fast is a nonintuitive technique: "failing immediately and visibly" sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production.

– Shore (2004)

The general rule when an anomaly is detected is to throw and exception. The alternative, i.e., returning some default value, while it allows the program continue, it may lead to malfunctions that that much more difficult to diagnose.

Most standard library methods use `Objects.assertNotNull()` to sanitize the input parameters and throw an exception in case they are `null`

8.8. Wrap-up

- Exceptions provide a mechanism to manage anomalies and errors
- Allow separating "nominal case" code from exceptional case code
- Decouple anomaly detection from anomaly handling
- They are used pervasively throughout the standard Java library
- Exceptions are classes extending the `Throwable` base class
- Inheritance is used to classify exceptions
  - `Error` represent internal JVM errors
  - `RuntimeException` represent programming error detected by JVM
  - `Exception` represent the usual application-level error
- Exception must be dispatched by
  - Catching them with `try{ }catch{ }`
  - Relaying with `throws`
  - Catching and re-throwing
- Unchecked exception can avoid mandatory dispatching ans simplify code
  - All exceptions extending `Error` and `RuntimeException` are unchecked
- The `finally` blocks can be used to execute some code in any possible case
  - the `try-with-resource` construct makes is easier to use them

9. Generics

Often in large applications, the same operations must be performed on objects of unrelated classes. A typical solution is to use `Object` references to accommodate objects of any type.

Let us consider the problme of representing pairs of values different types (e.g. `int`, `String`, etc.). If we use `Object` as universal reference type, a possible solution is:

Listing 9.1 Class Pair, Object-based version

```
public class Pair {
    Object a, b;
    public Pair(Object a, Object b ) {
    this.a=a;
    this.b=b;
    }
    Object first(){ return a; }
    Object second(){ return b; }
}
```

Of course the use of `Object` allows any type, provided it is a class type. Therefore primitives will be handled through wrapper classes.

An example of usage of the above Object-based solution

```
Pair sp = new Pair("One", "Two");                                    ①
Pair ip = new Pair(1,2);

String a = (String) sp.second();                                     ②
int i = (Integer) ip.first();

String b = (String) ip.second();                                     ③

Pair mixpair = new Pair(1,"Two");                                    ④
Pair pairmix = new Pair("One",2);

Object o = mixpair.second();                                         ⑤
if(o instanceof Integer i){

}
```

① `Object` allows usage with diverse types

② though you need explicit down-casts
③ down-casts are checked at run-time and my result in a `ClassCastException`
④ no check is possible at compile time about homogeneity of elements
⑤ thus extra code is required for safety

While `Object` references enable great flexibility, their use brings cumbersome code:

- several explicit casts are required,
- compile-time checks are limited
- down-casts can be checked at run-time only

A possible solution is to use generic types, which are implemented in Java using the *generics syntax*.

9.1. Generic Types

The *generic* alternative to the previous version (Listing 9.1) is shwon in Listing 9.2 below.

Listing 9.2 Class Pair, generics version

```
public class Pair<T> {
    private T a, b;
    public Pair(T a, T b) {
        this.a = a;
        this.b = b;
    }
    public T first(){
        return a;
    }
    public T second(){
        return b;
    }
    public void set1st(T x){
        a = x;
    }
    public void set2nd(T x){
        b = x;
    }
}
```

The generic declaration includes the `<T>` type parameter declaration. The generic class has `T` whenever that object-based version had `Object`

The generics version of the class can be used as follows:

```
Pair<String> sp = new Pair<>("Four", "Two");                    ①
Pair<Integer> ip = new Pair<>(4, 2);
```

```
Pair<String> mixp = new Pair<>(4, "Two");                                    ②

String a = sp.second();                                                      ③
int b = ip.first();
String bs = ip.second();                                                     ④
```

① Declaration is slightly longer since the actual type must be provided
② Compiler is able to check types are compatible
③ Use is more compact and safer since methods return the right type and auto-unboxing can be triggered
④ Compiler checks type matching

Use of generics leads to code that is:

- safer
- more compact
- easier to understand
- equally performing

9.1.1. Generic type syntax

Generic types can be declared using the following syntax:

```
(class|interface) Name <P1 {,P2, ...}>
```

- types can be either `class` or `interface`
- type parameters, `P1`, `P2` ...: represent types (classes or interfaces)

The type parameter are usually indicated with single uppercase letters. More in detail the convention is

- `T`(ype), for the general case of a type
- `R`(eturn), when the type represents the return type
- `E`(lement), when the type represents the type of elements withing a container
- `K`(ey), when the type represents the key type in a map or dictionary ADT
- `V`(alue), when the type represents the value type in a map or dictionary ADT

Reference type parameter must match the class parameter used in instantiation, e.g.

```
Pair<String> p=new Pair<String>("a","b");                                    ①
Pair<String> q=new Pair<>("a","b");                                          ②
```

① complete declaration for both variable declaration and instantiation
② the **diamond operator <>** lets the compiler infer the type in the instantiation

The *diamond operator* is available since Java 7.

9.1.2. Generic Library Interfaces

All standard interfaces and classes have been defined as generics since Java 5.

9.1.2.1. Generic `Comparable`

The generic `Comparable` interface is defined as follows

```
public interface Comparable<T>{
  int compareTo(T obj);
}
```

Without generics the implementation of `Comparable` looks like:

```
public class Student implements Comparable{
  int id;
  public int compareTo(Object o){
    Student other = (Student)o;                                    ①
    return this.id - other.id;
  }
}
```

① a cast is required

With generics the above code becomes:

```
public class Student implements Comparable<Student> {
  int id;
  public int compareTo(Student other){
    return this.id - other.id;
  }}
```

9.1.2.2. Generic `Comparator`

Similarly, the `Comparator` interface is defined as follows:

Listing 9.3 Comparator interface

```
public interface Comparator<T>{
  int compare(T a, T b);
}
```

Thus defining a comparator is very easy, e.g. for sorting strings by length:

```
Comparator<String> csl = (s1,s2) -> s1.length() - s2.length();
```

9.1.2.3. Generic `Iterator`

The generic `Iterable` and `Iterator` interfaces are defined as follows:

```
public interface Iterable<E>{
  Iterator<E> iterator();
}
```

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
}
```

Note the use of letter `E` since iterators are meant to browse the elements in a container.

An example of usage of the two interfaces to iterate on a sequence of random numbers can be rewritten as:

Listing 9.4 Iterable class RandomSeq, generic version

```
class RandomSeq implements Iterable<Double> {
    private double[] values;
    public RandomSeq(int n){
        values = new double[n];
        for(int i=0; i<n; ++i)
            values[i] = Math.random();
    }

    public Iterator<Double> iterator() {
        return new Iterator<Double>(){
            private int next=0;
            public boolean hasNext() {
                return next < values.length;
            }
            public Double next() {
                return values[next++];
            }
        };
    }
}
```

The usage of such classe is then very simple and readable:

With generics:

```
Random seq = new Random(10,5,10);
for(double v : seq){
  System.out.println(v);
}
```

Especially when compared to the one using the non generic version:

```
Random seq = new Random(10,5,10);
for(Object e : seq){
  double v = ((Double)e).doubleValue();
  System.out.println(v);
}
```

9.2. Generic Methods

9.2.1. Example

Method that need to operate on any type of element usually use `Object` typed arguments. For instance the following method removes a given element from an array then shift left the element to the right and filling with `null`

```
public static
Object remove(Object[] ary, Object el){
    for(int i=0; i<ary.length; ++i){
        if(ary[i]!=null && ary[i].equals(el)){
            Object removed = ary[i];
            ary[i] = null;
            return removed;
    }   }
   return null;
}
```

The above method can be used as follows:

```
String[] words = { "There", "must", "be", "some", "way", "out", "of", "here" };
Integer[] numbers = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
String f = (String)remove(words,"some");                                    ①
Integer i = (Integer)remove(numbers,13);

String oo =(String)remove(numbers,8);                                       ②
```

① down-casts are required to convert to the right type
② down-casts are checked at run-time and can result in `ClassCastException`

The alternative generic method can be written as

```
public static <E>                                                    ①
E remove(E[] ary, E el){                                             ②
    for(int i=0; i<ary.length; ++i){
        if(ary[i]!=null && ary[i].equals(el)){
            E removed = ary[i];
            ary[i] = null;
            return removed;
        }
    }
    return null;
}
```

① the type parameter `<E>` represents the type of the array elements
② `Object` is replaced by `E`

The code using this new version does not require any down-cast.

```
String[] words = { "There", "must", "be", "some", "way", "out", "of", "here" };
Integer[] numbers = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };

String f = remove(words,"some");                                     ①
Integer i = remove(numbers, 13);
String oo = remove(numbers, 8);                                      ②
String t = remove(words,8);
```

① down-cast are not required since the type are correct
② compiler checks type compatibility and issue error when *cannot convert*

9.2.2. Generic method syntax

The syntax for declaring a generic type is as follows

```
modifiers <P1 {,P2 ... }> type name( arguments )
```

The parameter types are declared before the return type.

The method arguments can be:

- primitive types
- regular types, i.e. classes and interfaces
- a type parameter, e.g. `P1`, `P2` etc.
- a generic type parameterized with the method's type parameter, e.g. `type<T>`

9.3. Type Variance

We must be careful about inheritance when generic types are involved.

In general if `A` extends `B` then for the generics instantiate with such actual type arguments, the language can implement

- **Covariance**: elements inheritance implies containers inheritance, i.e.

$$A \texttt{ extends } B \implies Container < A > \texttt{ extends } Container < B >$$

  This is safe for reading but unsafe for writing elements of the container

- **Contravariance**: elements inheritance implies inverse containers inheritance, i.e.

$$A \texttt{ extends } B \implies Container < B > \texttt{ extends } Container < A >$$

  This is safe for writing but unsafe for reading elements in the container.

- **Invariance**: elements inheritance does not imply container inheritance, this option guarantees type safety in any case but is is less flexible!

In Java, generics types are **invariant**. For instance, fact `Integer` extends `Object` does not imply `Pair<Integer>` extends `Pair<Object>`. Invariance leads to compilation errors:

```
Pair<Integer> pi = new Pair<>(0,1);
Pair<Object> pn = pi;
```

1. compiler cannot convert due to generics type invariance

To understand better the alternatives. If Java had implemented covariance for generics (as it does for arrays), it would results in type clashes as a consequence of writing in container.

```
Pair<Integer> pi = new Pair<>(1,2);
Pair<Number> pn = pi;                    ①
Number o=pn.first();                     ②
pn.set1st(1.0);                          ③
Integer i=pi.first();                    ④
```

① This assignment would be allowed if generic types were covariant
② reading a value from the container is ok
③ writing a Double into an Integer pair is a problem
④ this statement considered correct by the compiler, at run-time generates an error

On the other hand, if generics were contravariant, it would lead to type clashes as a consequence of reading from container

```
Pair<Number> pn = new Pair<>(1.0,2.0);
Pair<Integer> pi=pn;                                    (1)
pn.set1st(Integer.valueOf(42));                         (2)
Integer i = pi.second();                                (3)
```

(1) This assignment would be allowed if generic types were contravariant
(2) writing a value to the container is ok
(3) this statement considered correct by the compiler, at run-time generates an error

It is important to remember that differently from generics, Java considers arrays covariant.

Wrongly assuming co-variance could lead to an attempt to write a universal print method for pairs like the following one:

```
void printPair(Pair<Object> p) {
  System.out.println(p.first()+ "-" + p.second());
}
```

Unfortunately such a method won't work with, e.g. Pair,

```
Pair<Integer> p = new Pair<>(7,4);
printPair(p);                                           (1)
```

(1) the compiler cannot convert due to generics invariance

A more suitable way to write a universal method is to make it generic:

```
<T> void printPair(Pair<T> p) {
  System.out.println(p.first()+ "-" + p.second());
}
```

Even if declared as generic, the method in practice is not generic, the type `T` is never mentioned in the method. The compiler treats it like an `Object` and uses the relative `toString()` method to perform the concatenation.

9.4. Wildcards

The problem with regular generic methods is that type checking is present but it can be by-passed:

```
<T> void resetPair(Pair<T> p) {
  Object zero = Integer.valueOf(0);                     (1)
  p.set1st((T)zero);
  p.set2nd((T)zero);
}
```

(1) the compiler issues a warning of an unchecked cast but still accepts the code

The problem is that the above code could lead to an usafe code:

```
Pair<String> ps = new Pair<>("one","two");
resetPair(ps);                                                    ①
String s = ps.first();                                            ②
```

① the method with the "wrong" cast is executed without any problem
② a `ClassCastException` is thrown here

The exception is raised later and not where the actual forced cast was done in `resetPair()` method.

Sometimes when a method is formally generic but just for the purpose of being universal and thus it should not be using the type parameter in any way since it is as if it were unknown. It is possible to explicitly express this condition using the wildcard: `?`. The wildcard is usually pronounce as *unknown*.

```
void resetPair(Pair<?> p) {
  Object zero = Integer.valueOf(0);                               ①
  p.set1st(zero);
  p.set2nd(zero);
}
```

① the compiler issues an error since it cannot cast `Object` to an *unknown* type

In this case the more strict checks of the compiler forbid creating a method that performs any unsafe operation.

The syntax of the wildacard does not required declaring any type parameter since it is treated ad *unknown*.

The `?` (unknown) type is literally unknown therefore the compiler treats it in the safest possible way:

- Only methods from `Object` are allowed
- Assignment to an *unknown* reference is considered illegal

9.5. Bounded Types

The type parameters used in generics (types and methods) are **unbounded** by default, i.e. there are no constraints on the types that can be substituted for them.

The safe assumption for any type parameter `T` is that `T` extends `Object`. When the compiler checks the type usage, references of a type parameter `T` are considered to be able to provide the members that are defined in class `Object` (e.g., `equals()`). No other method can be called since at compile time the actual type of `T` is unknown.

This can be a problem since sometimes the methods implement algorithm the require specific behaviors from the objects. For instance, to find the max value in an array:

```
public static <T> void max(T ary[]){
    T max = null;
    for(T current : ary){
        if( max==null || ((Comparable)max).compareTo(current)<0){                    ①
            max = current;
        }
    }
    return max;
}
```

① the method `compareTo()` is not defined for type `T`, that is assumed to be `Object`

It is possible to possible to provide a constraint on a generic parameter, making it a **bounded type**

```
public static <T extends Comparable> void max(T ary[]){
    T max = null;
    for(T current : ary){
        if( max==null || max.compareTo(current)<0){                                  ①
            max = current;
        }
    }
    return max;
}
```

① the compiler assumes `T` extends `Comparable` thus allows calling `CompareTo()`

Upper bounds express constraints on type parameter's base class with the following syntax:

```
<T extends B { & B2 & ... } >
```

this means that at compile time, class `T` can be replaced only with types extending `B` – including `B` itself – and the other bounds, `B2`, … In this case `B` is called the *upper bound*

Lower bounds express constraints on type parameters's derived classes. Lower bounds can be provided only for unknown type.

The possible syntax for using bounded wildcards are:

- `<?>` unknown, unbounded
- `<? extends B>` upper bound: only sub-types of `B`, including `B`
- `<? super D>` lower bound: only super-types of `D`, including `D`

For instance to allow a conversion, a bounded wildcard must be used

```
double sumUB(Pair<? extends Number> p){
  return p.first().doubleValue()+p.second().doubleValue();                           ①
}
```

① compiler considers `p` as a `Pair<Number>` thus allows calling `doubleValue()`

9.6. Type Erasure

The implementation of generic types in Java aims at simplicity and efficiency. For each generic type declaration the compiler generates only one class. The generated class is obtained from the generic type by erasing the type parameters.

The classes corresponding to generic types are generated through a procedure called **type erasure**, the resulting class is called a **raw type**. The name of the raw type of generic class `G<T>` is `G`, i.e. the class name without the type parameters.

In general, each reference to the parameters is substituted with the parameter erasure. Erasure of a parameter is the erasure of its first boundary If no boundary is provided then the erasure is `Object`:

- For: `<T> T -> Object`
- For: `<T extends Number> -> T -> Number`
- For: `<T extends Number & Comparable> -> T -> Number`

9.6.1. Erasure consequences

Since there is only one class, within the methods there is no information available about the type parameters, in fact they have been erased. The actual value of type parameter is known only to the user of a generic type. Compiler applies checks only when a generic type is used, not within it.

Whenever a generic or a parameter is used a cast is added to its erasure. To avoid inconsistencies and wrong expectations:

- `instanceof` cannot be used on generic types
- `instanceof` is valid for `G` or `G<?>`

It is not possible to instantiate an object of the type parameter from within the class

```
class Triplet<T> {
        private T[] triplet;
        Triplet(T a, T b, T c){
            triplet = new T[]{a,b,c};
        }
}
```

The type parameter cannot be used in any way inside the raw type

Overloads and ovverrides are checked by compiler after type erasure

```
class Example<T, U> {
  void m(T t, U u){}                                    ①
  void m(U u, T t){}                                    ②
  void mm(T t){}                                        ③
  void mm(U u){}                                        ④
     U mm(T t){}                                        ⑤
}
```

① the erasure of both `T` and `U` is `Object`
② the erasure of both `T` and `U` is `Object`, so this is a duplicate of the previous
③ the erasure of `T` is `Object`
④ the erasure of `U` is `Object`, so this is a duplicate of the previous
⑤ the erasure of both `T` and `U` is `Object` but signature does not include the return type, so this is a duplicate of the two previous

Inheritance together with generic types leads to several possibilities, in particular it is not possible to implement twice the same generic interface with different types

Given a base class implementi `Comparable<T>`

```
class Student implements Comparable<Student> {..}
```

The derived class implementing the same interace:

```
class MasterStudent extends Student
        implements Comparable<MasterStudent> { .. }
```

Although the two interface differ in the type parameter, their erasure is the same, therefore it is considered a second implementation of the same interface.

9.6.2. Type inference

Upon generic method invocation, the compiler infers the type argument. The inferred type is called the **capture**.

In general, it is possible to explicitly indicate the capture, using what is called a **type witness**, although often useless, e.g. `Arrays.<Student>sort(sv);`

Inference of the *capture* is based on both the target type and the argument type

9.7. Use of Generics in Practice

9.7.1. Functional Interfaces

Predefined interfaces defined in `java.util.function` are all defined as generic types.

Specialized versions are defined for primitive types (int, long, double, boolean)

- Functions: `ToTypeFunction`, `Type1ToType2Function`
- Suppliers: `TypeSupplier`
- Predicate: `TypePredicate`
- Consumer: `TypeConsumer`

Listing 9.5 IntFunction functional interface

```
interface IntFunction<R> {
  R apply(int value);
}
```

For instance the `IntFunction` is defined as:

And it can be implemented using an instance method of object reference:

```
String hexDigits = "0123456789ABCDEF";
IntFunction<Character> hex = hexDigits::charAt;
System.out.println("Hex for 10 : " + hex.apply(10) );
```

It can be also implemented as a constructor method reference

```
IntFunction<Integer> builder = Integer::new;
Integer i = builder.build(1);
```

As another example, the `ToIntFunction` is defined as:

Listing 9.6 ToIntFunction functional interface

```
interface ToIntFunction<T> {
  int applyAsInt(T x);
}
```

And it can be implemented using an instance method reference

```
ToIntFunction<String> f = String::length;
for(String s : words){
  System.out.println(f.apply(s));
}
```

Functional interfaces provides also default and static factory methods that allow composing and building new implementations.

As far as the `Predicate` interface is concerned, the composition methods are

- `default Predicate<T> and(Predicate<T> o)`
- `default Predicate<T> or(Predicate<T> o)`
- `default Predicate<T> negate()`

As an example, the method `negate()` can be defined as follows:

The `Function` interface provides:

Listing 9.7 Implementation of Predicate.negate()

```
default Predicate<T> negate(){
    return o -> !this.test(o);
}
```

- `default <V> Function<T,V> andThen(Function<R,V> after)`
- `default <V> Function<V,R> compose(Function<V,T> before)`

As an example, the method `compose()` can be defined as follows:

Listing 9.8 Implementation of Function.compose()

```
default <V> Function<V,R> compose(Function<V,T> before){
    return x -> this.apply(before.apply(x));
}
```

9.7.2. Comparing

If we want to reorder the two elements in a pair so that the first is smaller or equal than the second we could write the following method that makes use of a `Comparator`:

```
<T> void sort(Pair<T> p, Comparator<T> cmp) {
    if(cmp.compare(p.first(),p.second()) > 0){
        T tmp = p.first();
        p.set1st(p.second());
        p.set2nd(tmp);
    }
}
```

The method can be used as follows

```
Pair<Integer> pi = new Pair<>(4,2);
Comparator ci=(i1,i2)-> i1.compareTo(i2);
Comparator<Number> cn = (n1,n2)->
                    Double.compare(n1.doubleValue(),
                                   n2.doubleValue());

sort(pi, ci);                                                    ①
sort(pi, cn);                                                    ②
```

① ok, the two actual arguments match the method declaration
② the method is not applicable for the argument: `Comparator<Number>` since a `Comparator<Integer>` was required

The compiler signals an error, but still it should be correct since a `Comparator<Number>` is capable of comparing also `Integer` objects.

The solution is to use an unknown with a lower bound:

```
<T> void sort(Pair<T> p, Comparator<? super T> cmp) {
    if(cmp.compare(p.first(),p.second()) > 0){
        T tmp = p.first();
        p.set1st(p.second());
        p.set2nd(tmp);
    }
}
```

With this version, `sort(pi, cn)` is correct because the compiler makes the following captures:

- `T` : `Integer`
- `?` : `Number` which is a *super* of `Integer`

According to this approach we can conclude that the `sort()` method of `Arrays` that accepts comparable objects should be defined as:

```
static <T extends Comparable<? super T>> void sort(T[] list);
```

Actually, for backward compatibility reasons, in class `Array` the methdo is defined as:

```
public static void sort(Object[] a)
```

As a consequence, no compile time check can be performed.

On the contrary, as expected, the `sort()` method that accepts a `Comparator` object is effectively defined as follows:

```
static <T> void sort(T[] a, Comparator<? super T> cmp)
```

So, in this case the compiler can perform the required checks and ensure correct behavior at run-time.

9.7.3. Generic Comparators

Interface java.util.Comparator

Listing 9.9 Comparator interface

```
public interface Comparator<T>{
  int compare(T a, T b);
}
```

```
Arrays.sort(sv, (a,b) -> a.id - b.id );
```

```
Arrays.sort(sv, new Comparator<Student>(){
    public void compare(Student a, Student b){
  return a.id - b.id
});
```

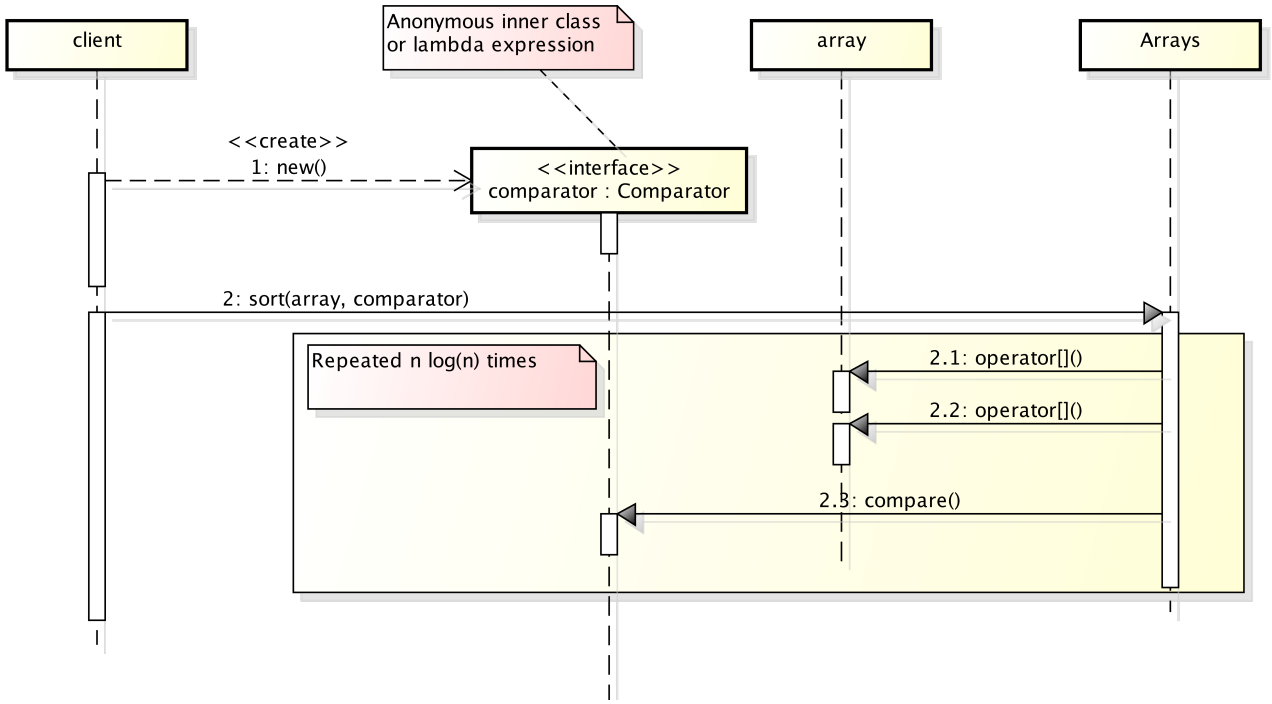


Figure 9.1.: Comparator behavior

Comparator factory

Most comparators take some information out of the objects to be compared Typically through a getter Such values are primitive or are comparable using their natural order (i.e. Comparable) Such comparator can be generated starting from a key getter functional object:

```
static <T,U extends Comparable<U>>
Comparator<T> comparing(Function<T,U> keyGetter)
```

Comparator.comparing

```
Arrays.sort(sv,comparing(Student::getId));
```

Requires `import static java.util.Comparator.*`

```
static <T,U extends Comparable<? super U>>
Comparator<T> comparing(Function<T,U> keyGetter){
  return (a,b) -> keyGetter.apply(a).
                       compareTo(keyGetter.apply(b));
}
```

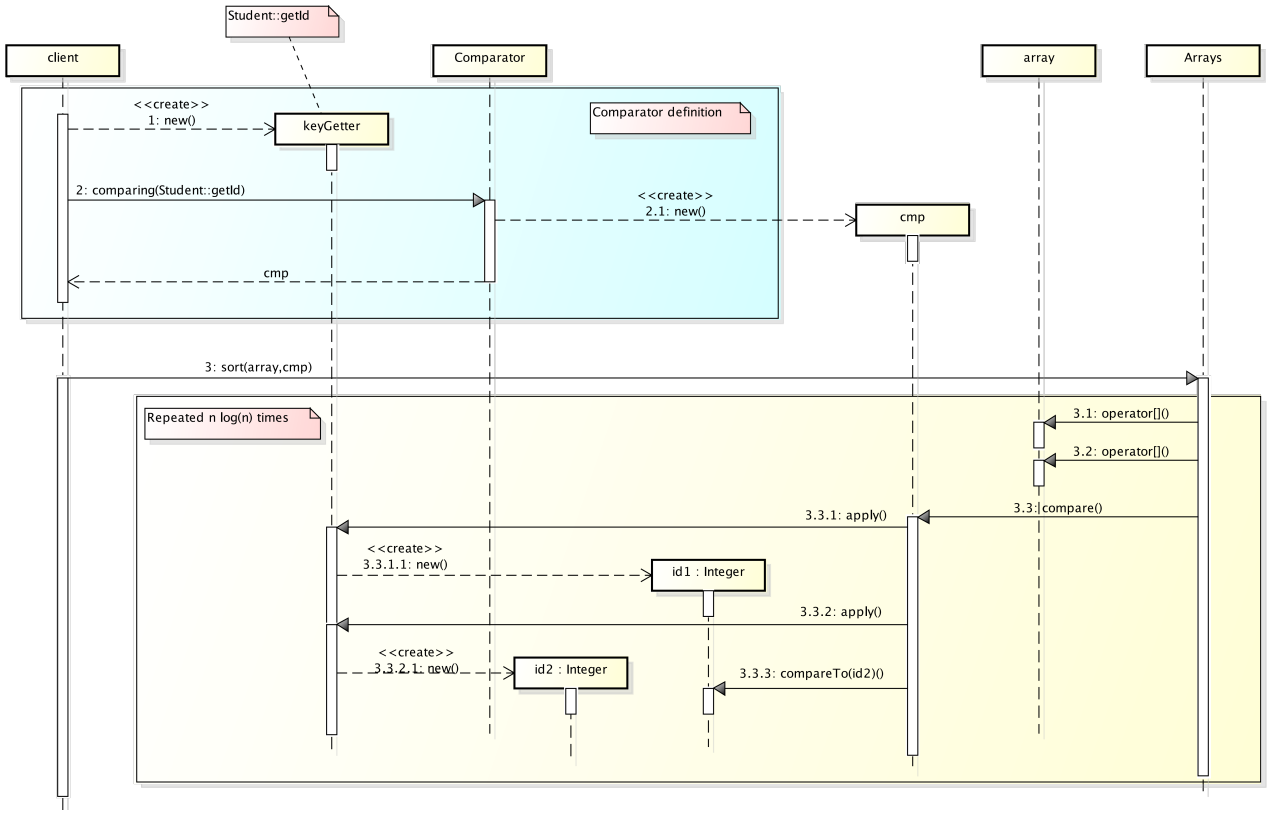


Figure 9.2.: Comparator factory behavior

Comparator.comparingInt

```
Arrays.sort(sv,comparingInt(Student::getId))
```

```
static <T,U extends Comparable<? super U>> Comparator<T>
 comparingInt(ToIntFunction<T,U> keyGetter){
  return (a,b) -> keyGetter.applyAsInt(a) -                                    keyGetter.applyAsInt(
}
```

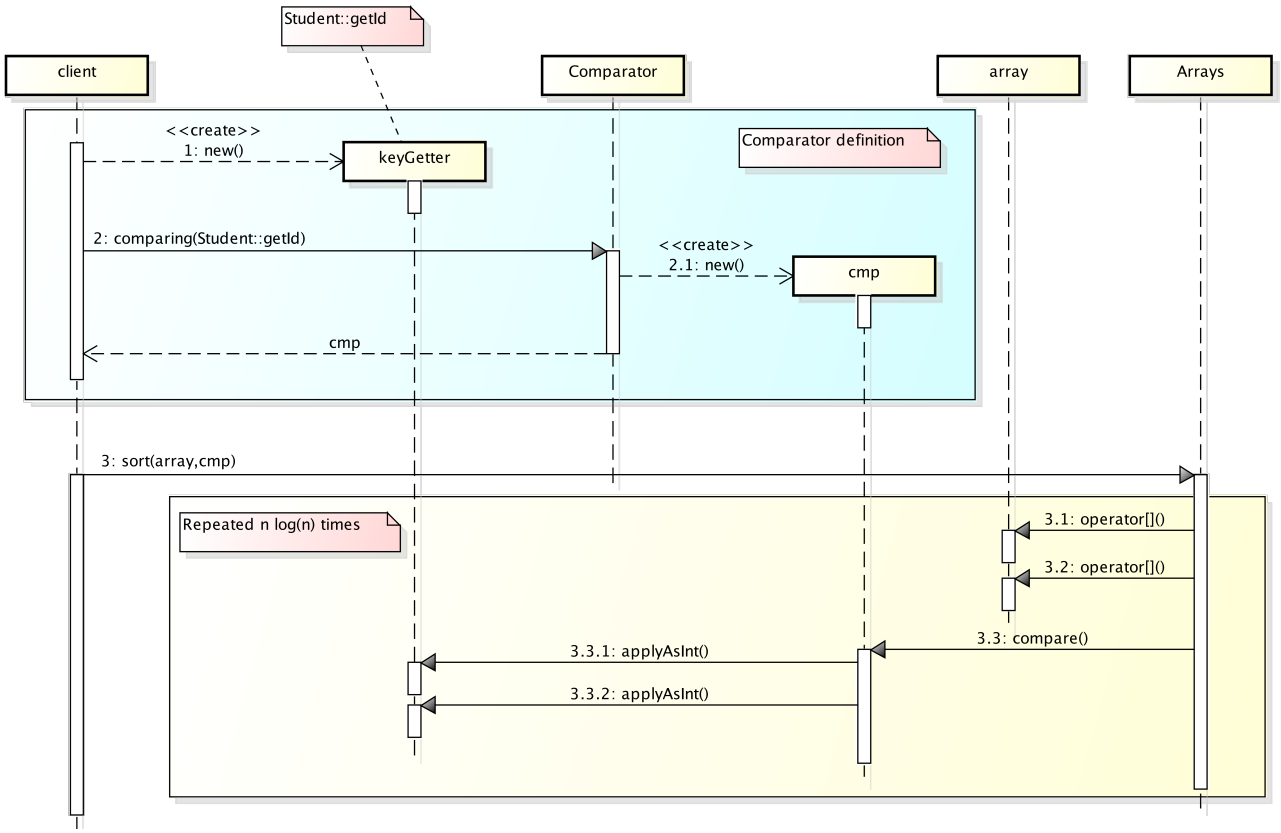


Figure 9.3.: IntComparator factory behavior

Performance

Table 9.1.: Relative performance of different comparator

Comparator	Time
`(a,b)-> a.id - b.id`	100
`(a,b)-> a.getId() - b.getId()`	120
`comparingInt(Student::getId)`	135
`comparing(Student::getId)`	185

It is interesting to look at a summary of how `Comparator` evolved during the history of Java.

Since Java 2 (1998):

```
Arrays.sort(sv,new Comparator(){
 public int compare(Object a, Object b){
    return ((Student)a).id-((Student)b).id;
 }});
```

Since Java 5 (2005)

```
Arrays.sort(sv,new Comparator<Student>(){
 public int compare(Student a, Student b){
    return a.getId() - b.getId();
 }});
```

Since Java 8 (2014), lambda expression:

```
Arrays.sort(sv,(a,b)->a.getId()-b.getId());
```

method reference

```
Arrays.sort(sv,comparing(Student::getId));
```

Functional interface composition Reverse order method Not a Comparator method!

```
static <T> Comparator<T>
  reverse(Comparator<T> cmp){
        return (a,b) -> cmp.compare(b,a);
}
```

```
Arrays.sort(sv,reverse(comparing(Student::getId)));
```

Comparator composition Reverse order Default method Comparator.reversed()

```
default <T> Comparator<T> reversed(){
  return (a,b) -> this.compare(b,a);
}
```

```
Arrays.sort(sv, comparing(Student::getId).reversed());
```

Comparator composition Multiple criteria Default method `Comparator.thenComparing()`

```
default <T> Comparator<T>
            thenComparing(Comparator<T> other){
  return (a,b) -> {
            int r = this.compare(a,b);
            if(r!=0) return r;
            else return other.compare(a,b);
}
```

Comparator composition Multiple criteria

```
default <U extends Comparable<U>
Comparator<T> thenComparing(Function<T,U> ke){
  return (a,b) -> {
     int r = this.compare(a,b);
     if(r!=0) return r;
   return ke.apply(a).compareTo(ke.apply(b));
}
```

Comparator composition

```
Arrays.sort(sv,(a,b)-> {
        int l = a.last.compareTo(b.last);
        if(l!=0) return l;
        return a.first.compareTo(b.first);
}));
```

```
Arrays.sort(sv,
    comparing(Student::getLast).
    thenComparing(Student::getFirst));
```

9.8. Wrap-up

- Generics allow defining type parameter for methods and classes with `<T>` syntax
- The same code can work with several different types
- Primitive types must be replaced by wrappers
- Generics containers are type invariant
- Wildcard, `?` (read as *unknown*)
- Generics are implemented by type erasure
- Most checks are performed at compile time

Part III.

Java API

10. Collections Framework

The collections framework consists in a set of related classes contained in the package `java.util` that provide:

- Interfaces for several ADTs (Abstract Data Types)
- Classes implementating ADTs
- Algorithms (sort)

Originally the were designed to use `Object` as a universal reference type. Since Java 5 redefined as generic types, in a backward compatible way.

The main interfaces of the collections framework are reported in the top part of Figure 10.1.

The ADT interfaces can be divided into two main groups:

- group containers, that collect simple objects using different strategies,
- associative containers, that store an association between two objects, called *key* and *value*.

The corresponding implementation classes are summarized in the bottom part of Figure 10.1.

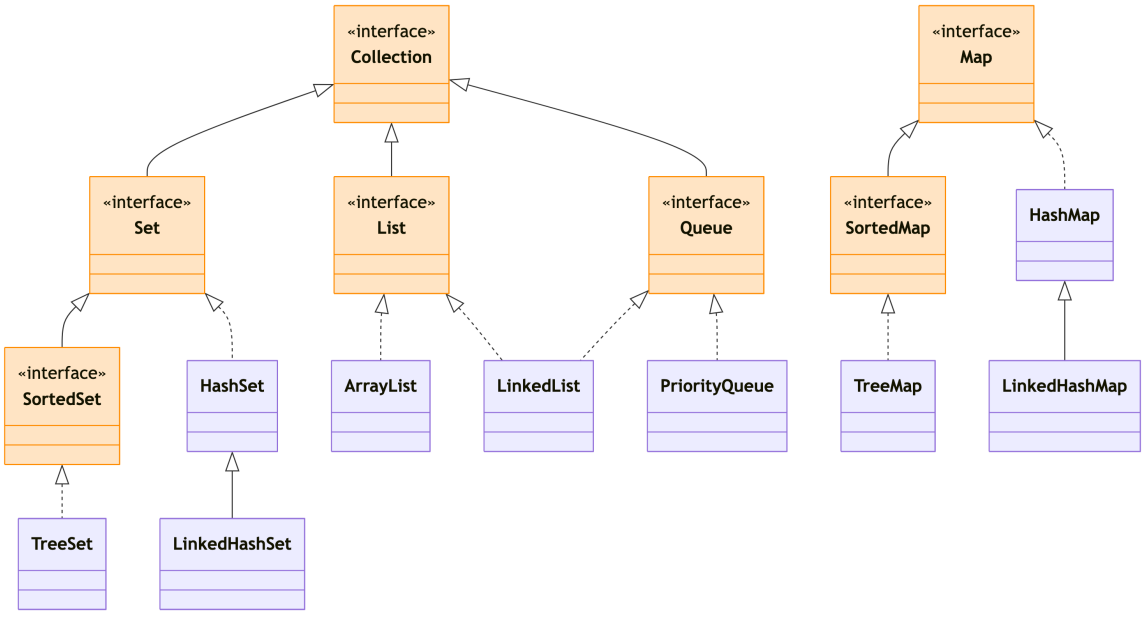


Figure 10.1.: Linked list structure example

Often for a single interface, multiple implementations are provided that leverage a different underlying data structure. The correspondance between interface, data structure, and implementation is summarize in Table 10.1.

Table 10.1.: Correspondence between data structure and collection implementation

↓ structure / interface →	Set	List	Map
Hash Table	`HashSet`		`HashMap`
Balanced Tree	`TreeSet`		`TreeMap`
Resizable Array		`ArrayList`	
Linked List		`LinkedList`	
Hash Table + Linked List	`LinkedHashSet`		`LinkedHashMap`

10.1. Group containers

10.1.1. Collection

The container represented by interface `Collection` represents a group of elements (references to objects) without any specific constraint. In particular it is not specified whether they are orderd or not, and whether the accept duplicates.

The interface `Collection` extends `Iterable` and thus can be used in the *for-each* construct.

All classes implementing `Collection` shall provide two constructors

- `C()`: creates an empty container
- `C(Collection c)`: creates a container filled in with the same elements present in `c`

The main methods are:

- `int size()`: number of elements present in the collection
- `boolean isEmpty()`: tests if the collection is empty, i.e. `size()==0`
- `boolean add(E element)`: add an element to the collection
- `boolean addAll(Collection<? extends E> c)`: add all elements in `c` to the collection
- `boolean contains(E element)`: tests if the given element in present in the collection
- `boolean containsAll(Collection<?> c)`: tests if all elements in `c` are present in the collection
- `boolean remove(E element)`: removes the given element from the collection, `true` if succeeds
- `boolean removeAll(Collection<?> c)`: removes all elements from `c` from the collection, `true` if succeeds
- `void clear()`: removes all elements from the collection
- `Object[] toArray()`: creates an array containing the elements present in the collection

The methdod `Object[] toArray()` returns an `Object` even if the interface uses generics because everywhere else. This is due to generic types limitations deriving from type erasure, the method cannot instantiate an array of the correct type.

As an alternative, the overload `<T> T[] toArray(T[])` accepts an array of the target type. If not large enough a new array of the same type and right length is instantiated and returned; it uses the `Arrays.copyOf()` method that create an array of the right type. In practice, an empty array of the correct type can be passed and let the method do the allocation.

The interface `Collection` has no direct implementations. Although it can be used as a more general reference type to specific container types, for instance:

```
Collection<Person> persons = new LinkedList<Person>();                    ①
persons.add( new Person("Alice") );                                       ②
System.out.println( persons.size() );                                     ③
Collection<Person> copy = new TreeSet<Person>();                          ④
copy.addAll(persons);                                                     ⑤
Person[] array= copy.toArray(new Person[0]);                              ⑥
System.out.println( array[0] );                                           ⑦
```

① instantiates a new collection, implemented by `LinkedList` class
② adds a new element to the collection
③ prints the size, the is `1` at this point
④ creates a new instance of a collection, implemented by `TreeSet` class
⑤ adds all elements in `persons` into `copy`, the two operations could be conflated into `new TreeSet<Person>(persons)`
⑥ creates an array containing the elements of the collection
⑦ prints the first (and only) element of the array

10.1.2. List

The container represented by interface `List` is a specialization of `Collection` – which it implements –. It can contain duplicate elements, in addition the insertion order is preserved and the user can define the insertion point, and most importantly the elements can be accessed by position. Extends Collection interface

The main methods of the `List` interface, in addition to those inherited from `Collection`, are:

- `E get(int index)`: retrieves the element at the given position
- `E set(int index, E element)`: replaces the element at the given position
- `void add(int index, E element)`: adds an element in a given position, can cause a shift of elements already present
- `E remove(int index)`: removes the element at the given position and returns it
- `int indexOf(E o)`: returns the index of the first occurrence of the given element
- `int lastIndexOf(E o)`: returns the index of the latest occurrence of the given element
- `List<E> subList(int from, int to)`: returns a list that contains a portion of the starting one

An example of usage of class `List` is:

```
List<Integer> l = new ArrayList<>();

l.add(42);                                                                ①
l.add(0, 13);                                                             ②
l.set(0, 20);                                                             ③
int a = l.get(1);                                                         ④
l.add(9, 30);                                                             ⑤
l.add(2, 30);                                                             ⑥
```

① 42 in position 0
② 42 moved to position 1

③ 13 replaced by 20
④ returns 42
⑤ Error: out of bounds, `IndexOutOfBoundsException: Index: 9, Size: 2`
⑥ Ok, immediately after the last element, same as `add()`

The main `List` implementations are:

- `ArrayList<E>` it is based on a resizable array,
- `LinkedList<E>` it is based on linked elements.

The two classes impelements the method defined in the interface and in addition provide additional methods that are convenient for the specific data structure used:

10.1.2.1. LinkedList

The `LinkedList` implementation uses a series of linked elements as shown in Figure 10.2. The list maintains a reference to the first element `first` and one to the last one `last`.

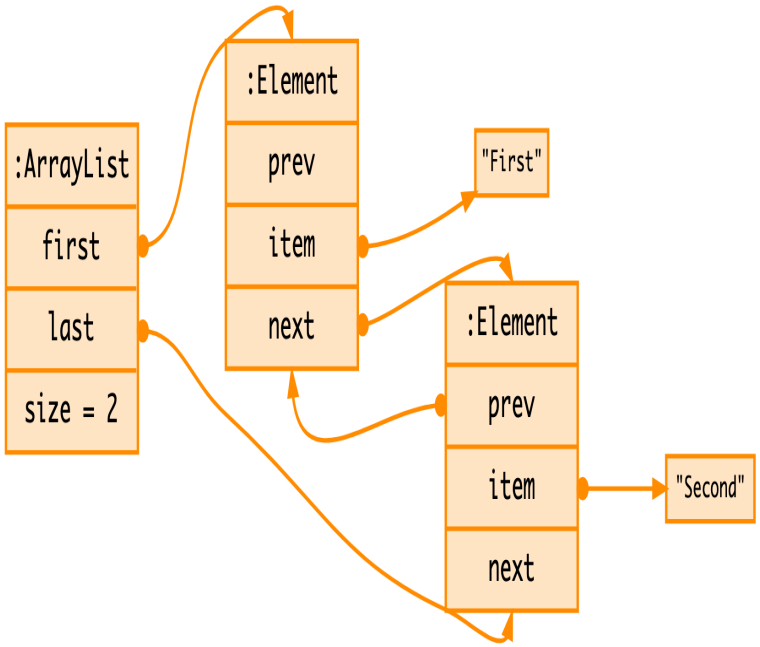


Figure 10.2.: Linked list structure example

The `add()` operations adds a new element at the tail of the list as shown in Figure 10.3.

Essentially:

- a new element is created, its next element being the last one,
- the next element of the last is the newly created element,
- the new element becomes the last.

The class `LinkedList<E>` provides the following additional constructors and methods that leverage the internal structure:

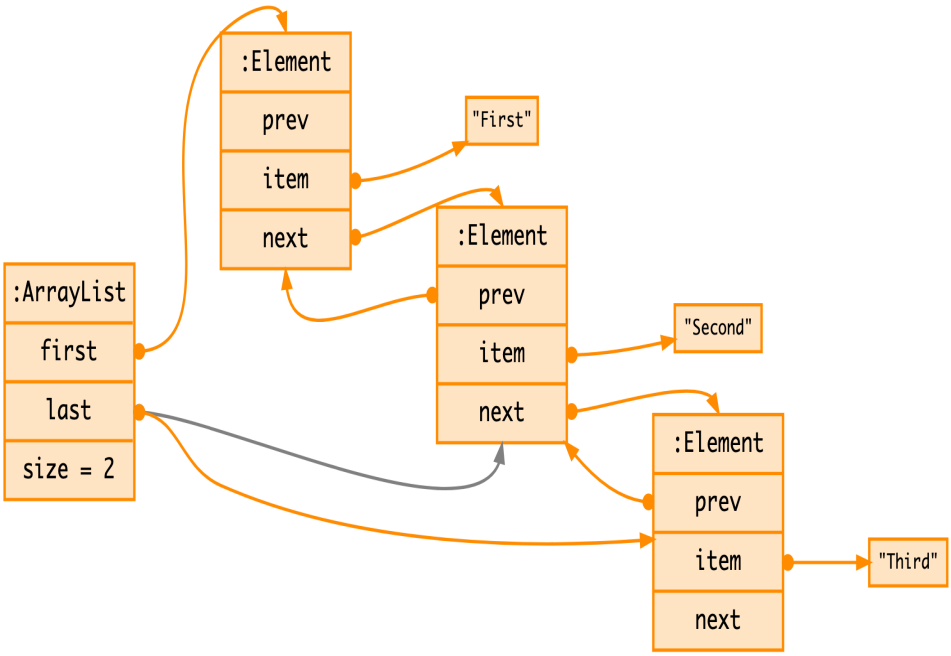


Figure 10.3.: Linked list structure example

- `void addFirst(E o)`: adds an element in first position, i.e. `l.add(0, o)`
- `void addLast(E o)`: adds an element in the latest position, i.e. `l.add(o)`
- `E getFirst()`: returns the first element, i.e. `get(0)`
- `E getLast()`: returns the last element, i.e. `l.get( l.size()-1 )`
- `E removeFirst()`: removes the first element
- `E removeLast()`: removes the last element

10.1.2.2. Array list

The `Arraylist` implementation uses an array that is resized when required. An example is shown in Figure 10.4.

The `add()` operations adds a reference to the element at the next available place in the array, as shown in Figure 10.5.

When the array is full a resize operation is required as shown in Figure 10.6.

In summary:

- a new (larger) array is created and the existing elements are copied into it (`Arrays.copyOf()`)
- then the first new position of the array will store the reference to the element

The operation of instantiating the new array and copying the elements is very expensive, for this reason to improve performance it can be advisable to define an initial size suitable for the usage.

The class `ArrayList<E>` provides the following additional constructors and methods:

- `ArrayList(int initialCapacity)`: initializes the underlying array to the given initial size,

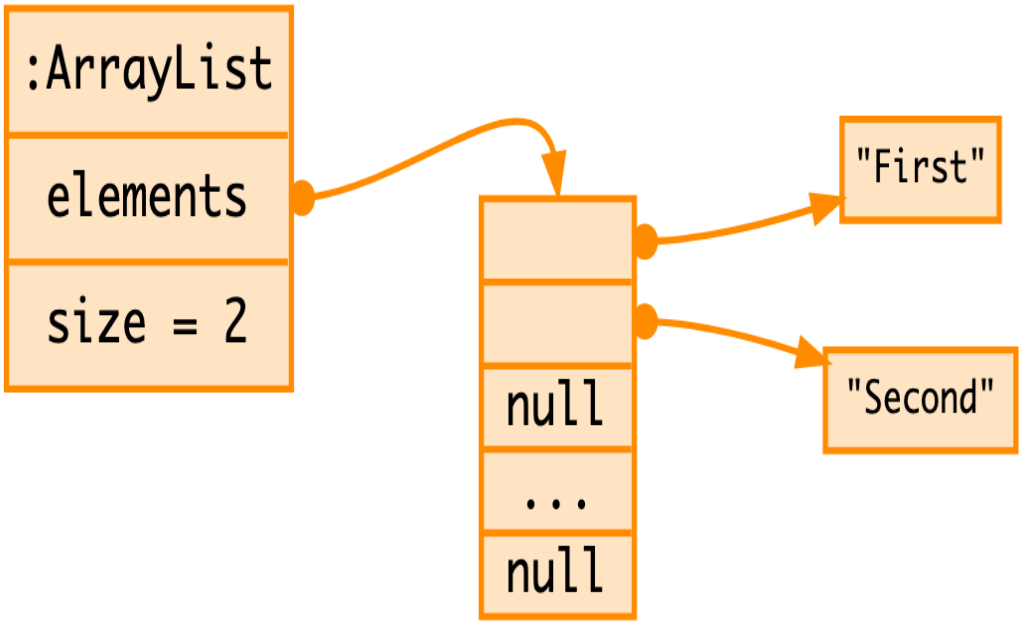


Figure 10.4.: Array list structure example

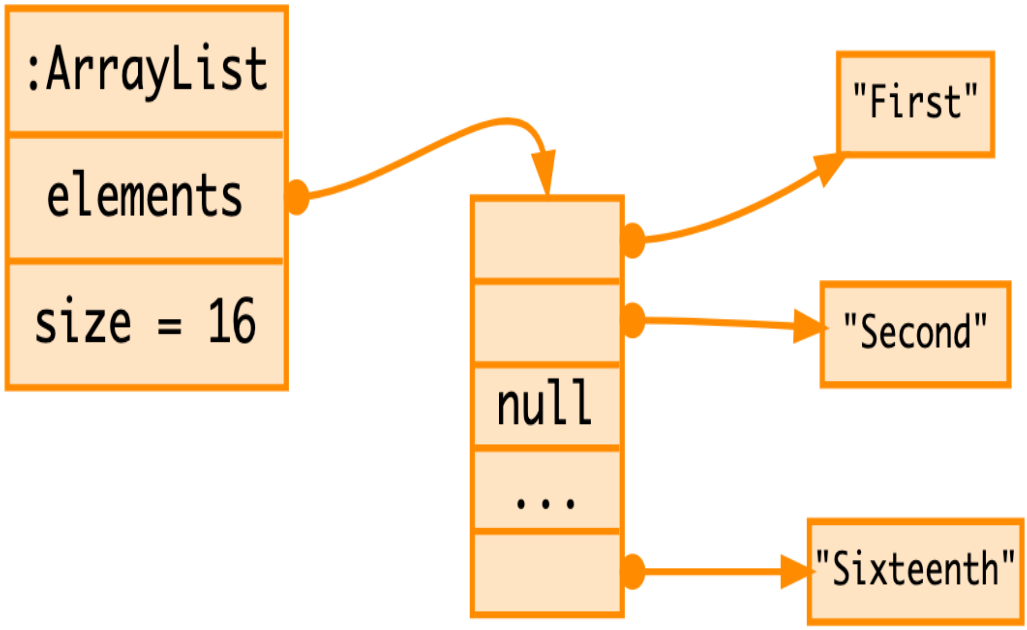


Figure 10.5.: Array list structure example

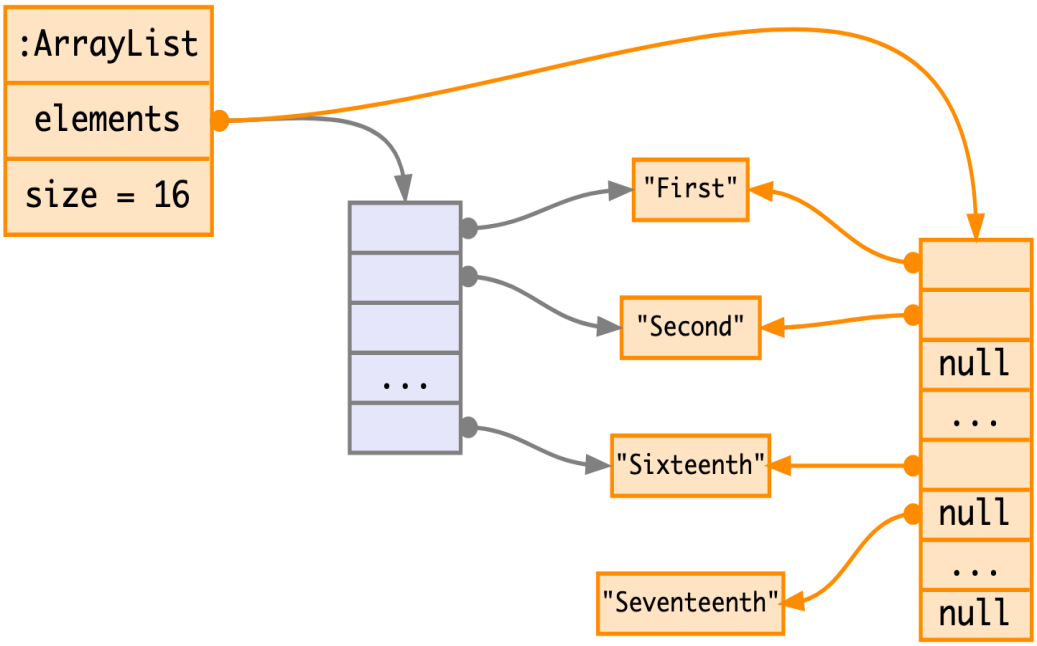


Figure 10.6.: Array list structure example

- `void ensureCapacity(int minCapacity)`: resizes the underlying array so that is has at least the given size.

10.1.3. Queue

The container represented by interface `Queue` is a specialization of the `Collection`. The elements are inserted in a specific order. In particular, it defines a head position where is the first element that can be accessed.

The typical orders are:

- insertion order, i.e. FIFO (Fist-In-First-Out), or
- elements' natural order (Priority queue).

Interface `Queue`, in addition to those inherited from `Collection`, defines the following main methods:

- `peek()`: just retrieves the element at the head of the queue
- `poll()`: retrieves and removes the element at the head of the queue

The `Queue` interface has two predefined implementations:

- `LinkedList`, in this case the head is the tail of the list, it is a FIFO;
- `PriorityQueue`, the head is the smallest element.

An example usage of `Queue` that compares the two predefined implementations:

```
Queue<Integer> fifo = new LinkedList<Integer>();                    ①
Queue<Integer> pq = new PriorityQueue<Integer>();
fifo.add(3); pq.add(3);                                             ②
fifo.add(1); pq.add(1);
fifo.add(2); pq.add(2);
System.out.println(fifo.peek());                                    ③
System.out.println(pq.peek());                                      ④
```

① create two queues using a FIFO and priority strategies
② adds the same elements in the same order to both queues
③ the head element of the `fifo` is the first added
④ the head element of the `pq` is the smallest

10.1.4. Set and SortedSet

The interface `Set` contains no additional methods w.r.t. those inherited from `Collection`. It serves the purpose of a flagging interface. It is intended to represents containers that have no duplicate elements. The duplication detection is demanded to the `equals()` method.

The classes implementing `Set` must comply with the following restriction on the `add()` method:

$$\forall e_1, e_2 \in \Sigma : e_1 \neq e_2 \implies e_1.\texttt{equals}(e_2) == \texttt{false}$$

The elements are traversed in no particular order

A further specialization of the `Set` interface is `SortedSet` that mandates that the elements are traversed according to the element ordering. Like its parent interface, it does not allow duplicate elements.

The interface `SortedSet` provides the following additional methods:

- `E first()`: retrieved the first element in order
- `E last()`: retrieved the last element in order
- `SortedSet<E> headSet(E toElement)`: retrieved the initial portion of elements in order until the given one
- `SortedSet<E> tailSet(E fromElement)`: retrieved the final portion of elements in order starting from the given one
- `SortedSet<E> subSet(E from, E to)`: retrieves a subset consisting of the elements in orderd comprised within the two provided

Depending on the constructor used, it is possible to use two alternative implementations of the custom ordering:

- `TreeSet()` default constructor implies natural ordering (elements must be implementations of `Comparable`),
- `TreeSet(Comparator c)` comparator constructor implies the ordering according to the comparator criteria.

There are three standard implementations:

- `HashSet` implements `Set`, usus hash tables as internal data structure (faster)
- `LinkedHashSet` extends `HashSet`, in addition the elements are traversed by iterator according to the insertion order
- `TreeSet` implements `SortedSet`, uses R-B trees as internal data structure (computationally expensive)

10.2. Associative containers

10.2.1. Map

The interface `Map` represents containers that associates keys to values (e.g., SSN -> Person). Keys and values must be objects, and keys must be unique. For a given key, only one value can be stored.

The following constructors are common to all map implementers:

- `M()`: creates an empyt map
- `M(Map m)`: creates a map containing the same entries as the given map

The `Map` interface provides the following main methods:

- `V put(K key, V value)`: add an entry `key-value` in the map
- `V get(K key)`: retrieves the value associated with the given `key`
- `V remove(K key)`: removes the association for the given `key`
- `boolean containsKey(K key)`: tests if the map contains the given `key`
- `public Set<K> keySet()`: returns the set of all keys present in the map
- `public Collection<V> values()`: returns the collection of all values present in the map
- `int size()`: returns the number of entries present in the map
- `boolean isEmpty()`: tests if the map is empty, i.e. `size()==0`
- `void clear()`: removes all entries from the map

Example of usage of the `Map` interface:

```
Map<String,Person> people =new HashMap<>();                                    ①
people.put( "ALCSMT",  new Person("Alice Smith") );                            ②
people.put( "RBTGRN",  new Person("Robert Green") );

if( ! people.containsKey(©`))                                                  ③
    System.out.println( "Not found" );

Person bob = people.get("RBTGRN");                                             ④

int populationSize = people.size();                                            ⑤

for(Person p : people.values()){                                               ⑥
    System.out.println(p);
}

for(String ssn : people.keySet()){                                             ⑦
```

```
    System.out.println(ssn);
}
```

① creates a new map implemented by a `HashMap` class
② adds two entries in the map
③ checks if the key `"RBTGRN"` is present in the map
④ retrieves the person associated to key `"RBTGRN"`
⑤ retrieves the number of entries in the map, 2
⑥ retrieves the values in the map, i.e. the persons
⑦ retrieves the set fo keys in the map

A furhter specialization of `Map` is represented by `SortedMap` that mandates that its elements be traversed according to the keys' ordering. The ordering can be based on the natural ordering (`Comparable`) or provided as a `Comparator` object passed to the constructor.

It provides the followint specialized methods:

- `SortedMap headMap(K toKey)`: creates a new sorted map the contains only initial keys, up to the given one
- `SortedMap tailMap(K fromKey)`: creates a new sorted map the contains only final keys, starting from the given one
- `SortedMap subMap(K fromKey, K toKey)`: creates a new sorted map the contains only keys within the two provided
- `K firstKey()`: retrieves the first key
- `K lastKey()`: retrieves the last key

The standard implementations of `Map` interface are

- `HashMap` implements `Map`, keys have no order, but must provide an hash code
- `LinkedHashMap` extends `HashMap`, key have no order, but values are traversed in insertion order
- `TreeMap` implements `SortedMap`, keys are traversed in ascending key order and values are traversed in ascending order of the relative key

10.2.1.1. HashMap

Class `HashTable` is based on an hash table data structure. The table is automatically resized when a specific load factor reached. By default the load factor is 0.75 and the initial size is 16. These values can be changed through a constructor. The class shares part of the implementation with the `HashSet` class, thus most consideration here apply also to that class.

The basic structure of the `HashMap` is exemplified in Figure 10.7.

Hash table based containers `HashMap` and `HashSet` work better if entries define a suitable `hashCode()` method. Values must be as spread as possible, otherwise, collisions occur. A collision occurs when two entries fall in the same bucket. In such a case elements are put in a chained list, chaining reduces time efficiency. Get and put operatiosn take constant time in case of no collisions.

The chaining procedure is shown in Figure 10.8.

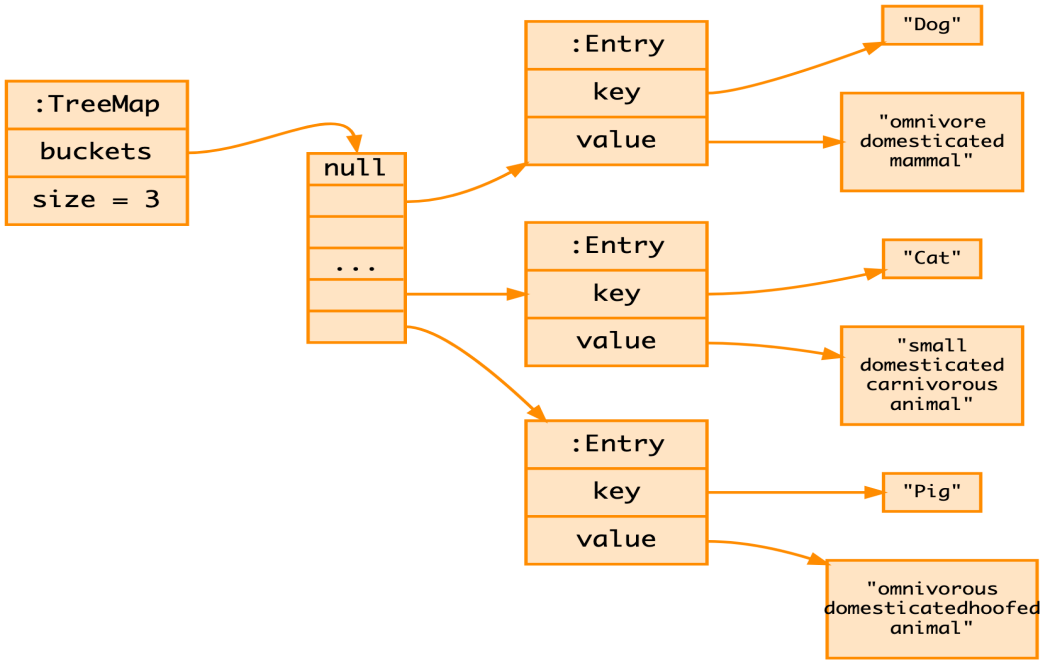


Figure 10.7.: HashMap internal structure example

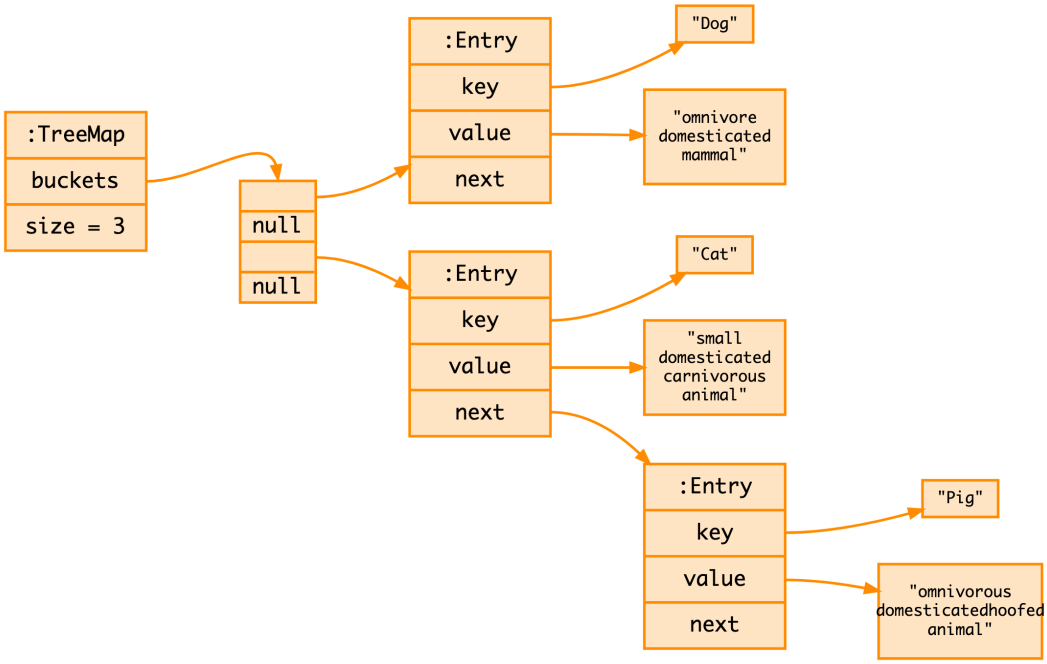


Figure 10.8.: HashMap internal structure example

10.2.1.2. TreeMap

The class `TreeMap` is based on a Red-Black tree. The get and put operations takes $log(n)$ time. Due to the structure of the tree, the keys are maintained and traversed in order. Therefore it is required that the key class (`K`) must be `Comparable` or a `Comparator` must be provided to the constructor.

An exemplification of the `TreeMap` structure is shown in Figure 10.9.

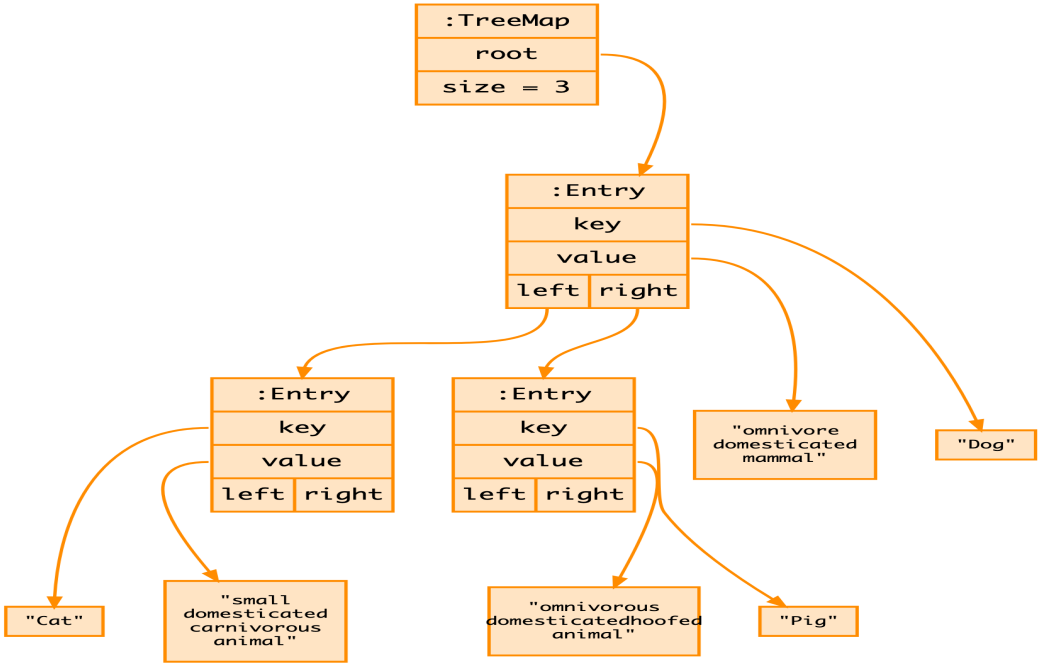


Figure 10.9.: TreeMap internal structure example

10.3. Optional

The typical convention in Java APIs is to let a method return a `null` reference to represent the absence of a result. This approach, require that the caller must check the return value of the method to detect that case. If the caller omits – forget? – such a check, a `NullPointerException` (NPE) may occur. This is called the **nullability problem**.

The collections framework contain class `Optional` that is intended to mitigate the nullability problem. Class `Optional` is used to represent a *potential* value, i.e. a value that might be `null`.

Methods returning `Optional<T>` make explicit that the return value may be missing. Such a return type forces the clients to deal with potentially empty optional.

Access to the value embedded within the `Optional` can be done through:

- `boolean isPresent()`: checks if `Optional` contains a value
- `T get()`: returns the value if present; otherwise it throws a `NoSuchElementException`.
- `ifPresent(Consumer<T> block)`: executes the given code if a value is present.

- `T orElse(T default)`: returns the value if present; otherwise it returns the provided `default` value.
- `T orElse(Supplier<T> s)`: returns the value if present; otherwise it returns the value supplied by `s`

The `Optional` objects can be created through a set of static factory methods:

- `of(T v)`: throws exception if `v` is `null`
- `ofNullable(T v)`: returns an empty `Optional` when `v` is `null`
- `empty()`: returns an empty `Optional`, same as `ofNullable(null)`

Such methods force the programmers to think about what they are about to return.

10.4. Using Collections

> **Note**
>
> The reccomendation is to always use the most general interface suitable for the usage in the program. E.g. `List<>` is better than `LinkedList<>`.

The motivations for such recommendation are:

- more general interfaces are more flexible for future changes, i.e. a change in the specific implementing class does not affect the usage, which always goes through the general interface.
- improve the programmer's approach, makes them think first about the type of container, then about the implementation

The general rule for selecting the container type is:

- if access by key is needed use a `Map`
  - if values sorted by key use a `SortedMap`
- otherwise use a `Collection` (or derived interfaces)
  - if indexed access, use a `List`
  - if access in order, use a `Queue`
  - if no duplicates, use a `Set`
  - if elements sorted, use a `SortedSet`

10.4.1. Using lists

10.4.1.1. Iteration

Since all group containers – through `Collection` – implements the `Iterable` interface, they can be used with the try-catch construct:

```
Iterable<Person> persons = new LinkedList<Person>();

for(Person p: persons) {
    System.out.println(p);
}
```

In addition `Iterable` defines the default method:

- `forEach(Consumer<? super T> action)` perfoms an action on each element.

The method can be used to perform operations of elements with a functional interface:

```
Iterable<Person> persons = new LinkedList<Person>();

persons.forEach( p -> {
    System.out.println(p);
});
```

It is important to note that it is unsafe to iterate over a collection while modifying – i.e., add or remove elements – at the same time:

```
List<Integer> lst=new LinkedList<>();
lst.add( 10 );
lst.add( 11 );
lst.add( 13 );
lst.add( 20 );

int count = 0;
for (Iterator<?> itr = lst.iterator(); itr.hasNext(); ) {
   itr.next();
   if (count==1)
      lst.remove(count);                                                    ①
   count++;
}
```

① a `ConcurrentModificationException` is raised at run-time

The correct way of modifying while iterating is by using the iterator's own methods:

- `Iterator.remove()`: removes the current element
- `ListIterator.add()`: adds an element at the current position

```
List<Integer> lst = new LinkedList<>();
lst.add( 10 );
lst.add( 11 );
lst.add( 13 );
lst.add( 20 );
```

```
int count = 0;
for(Iterator<?> itr = lst.iterator(); itr.hasNext(); ) {
   itr.next();
   if (count==1)
      itr.remove(); // ok
   count++;
}
```

10.4.1.2. Time Performance

Based on the implementation structured of the two list classes, the general time performance for the two main implementations of `List` are sumamrized in the following Table 10.2.

Table 10.2.: Summary of time performance behavior of main `List` implementations.

Operation	`ArrayList`	`LinkedList`
`get(n)`	Constant	Linear
`add(0,e)`	Linear	Constant
`add()`	Constant	Constant

More in detail we can build the following analytical models for two insertion methods:

`add(0,e)`: add in first position in list of size n:

- `LinkedList`: $t(n) = C_L$
- `ArrayList`: $t(n) = n\dot{C}_A$

`for(int i=0; i<n; ++i) l.add(i);`: add n elements in an initially empty list

- `LinkedList`: $t(n) = n \cdot C_L$
- `ArrayList`: $t(n) = \sum_{i=1}^{n} C_A \cdot i = \frac{C_A}{2} n \cdot (n-1)$

On a MacBookPro M2, $C_L = 16.0ns$ and $C_A = 0.2ns$

10.4.2. Using maps

10.4.2.1. Getting an item

The retrieval of an element from a map should consider the case of a missing element (i.e. key not present). A first option is to check the return value of `get()` that returns `null` if the key is not found:

```
String val = map.get(key);
if( val == null ) val = "Undefined";
```

The above solution has a possible vulnerability, if the value associated with a key is exactly `null` a valid return value cannot be distinguished from a missing key.

Alternatively it is possible to first check the presence of the key and then retrive the value:

```
if( ! map.containsKey(key))
     val = "Undefined";
else
     val = map.get(key);
```

Since this kind of operation is very common, a specific default method `getOrDefault()` has been added to the interface, which can be used for the same purpose:

```
String val = map.getOrDefault(key, "Undefined");
```

10.4.2.2. Updating entries

Another common operation is to update the entries, i.e. changing the value associated with a key. A typical exambe is when a map is used to count the occurrences of (repeated) items in a collection. A possible example is:

```
Map<String,Integer> wc=new HashMap<>();
for(String w : words) {
    Integer i= wc.get(w);
    wc.put(w, i==null?1:i+1);
}
```

Since this operation is very commong a suitable default function `compute()` is provided. The method `compute()` accepts two argument:

- the key,
- a `BiFunction` that takes the key and value present in the map (value is `null` if not present) and computes the new value to be stored.

A simple alternative to the above code for counting occurrence that uses `compute()` can be:

```
Map<String,Integer> wc=new HashMap<>();
for(String w : words) {
    wc.compute(w,(k,v)->v==null?1:v+1);
}
```

The above code although clean and compact is not particularly efficient since it uses the autoboxing feature.

Every increment of an Integer is equivalent to `Integer.valueOf(v.intValue()+1)`, that requires additional time for the instantiation and and additional memory fee of 16 bytes per object creation.

A much better solution would be to use a mutable object. For instance:

```
class Counter {
    private int i=0;
    public String toString(){
        return ":"+i;
    }
    public void inc(){ ++i; }
}
```

In this case another default method can be usefule `computeIfAbsent()` that allows creating a `Counter` objects only on the first occurrence of a word and just increment the counter on every occurrence:

```
Map<String,Counter> wc=new HashMap<>();
for(String w : words) {
    wc.computeIfAbsent(w, k->new Counter()).inc();
}
```

This solution is ~40% faster than the one using `Integer` and requires 16 bytes less per each increment.

10.4.2.3. Keeping items sorted

Using sorted maps

```
SortedMap<String,Integer> wc=new TreeMap<>();
```

produces:

```
`"A"=1, "All"=3, "And"=2, "Barefoot"=1,…`
```

Vs. using non sorted maps:

```
Map<…> wc=new HashMap<>();
```

produces:

```
"reason"=1, "been"=1, "spoke"=1, "let"=1
```

size	HashMap	TreeMap	ArrayList	LinkedList

10.4.2.4. Time performance

Example: 100k searches in a container with two different sizes, require

size	HashMap	TreeMap	ArrayList	LinkedList
100k	3ms	60ms	40s	>1h
200k	3ms	65ms	110s	

10.5. Algorithms

A number of common algorithms are provided as static methods of `java.util.Collections`. Please note the trailing `s`!

Most of the algorithms work on `List` since it has the concept of position:

- `sort()` - merge sort of `List`
- `binarySearch()` – requires ordered sequence
- `shuffle()` – unsort
- `reverse()` - requires ordered sequence
- `rotate()` – of given a distance
- `min()`, `max()` – min and max in a `Collection`

10.5.1. Sort

The method uses the merge sort algorithm that has a $n \log(n)$ time complexity. Requires a `List<E>` since it needs access by index.

As the `sort()` method in `Arrays`, it has two overloads:

- `<T extends Comparable<? super T>> void sort(List<T> list)`
- `<T> void sort(List<T> list, Comparator<? super T> cmp)`

10.5.2. Search

- `<T> int binarySearch(List<? extends Comparable<? super T>> l, T key)`
- `<T> int binarySearch(List<? extends T> l, T key, Comparator<? super T> c)`

Searches the specified object, the list must be sorted into ascending order according to natural ordering or comparator's order.

10.6. Wrap-up

- The collections framework includes interfaces and classes for containers
- There are two main families
  - Group containers
  - Associative containers
- All the components of the framework are defined as generic types

11. Java Stream

Stream: a sequence of elements from a source that supports data processing operations. Operations are defined by means of behavioral parameterization

11.1. Basic features:

The key features of the Stream API in Java are:

- Pipelining
- Internal iteration: no explicit loops statements
- Lazy evaluation (pull-architecture): no work until a terminal operation is invoked

11.1.1. Pipelining

Often processing of large amounts of information consists in several steps applied one after another.

There are several different approaches to writing such chained procesessing:

- using intermediate step results,
- using function composition,
- using a pipeline.

Let us consider the simple case of a processing chain that prints the first four elements of sequence of words after converting them to lower case.

Using intermediate steps results it can be written as:

```
List<String> words =  Arrays.asList("There","must","be","some","way","out","of","here");
List<String> lowercase = map(words, String::toLowerCase);
List<String> four = lowercase.subList(0, 4);
four.forEach(IO::println);
```

An alternative approach is the composition of functions:

```
forEach(
    limit(
        map(Arrays.asList("There","must","be","some","way","out","of","here"),
            String::toLowerCase
        ),
        4
```

```
    ),
    IO::println
);
```

Finally with the use of the Stream API it can be written as a pipeline:

Listing 11.1 Sample lyrics print with streams

```
Stream.of("There","must","be","a","way","out","of","here")
      .map(String::toLowerCase)
      .limit(4)
      .forEach(System.out::println);
```

Each stage in the expression, written one per line, can be represented as a pipeline as represented in Figure 11.1.

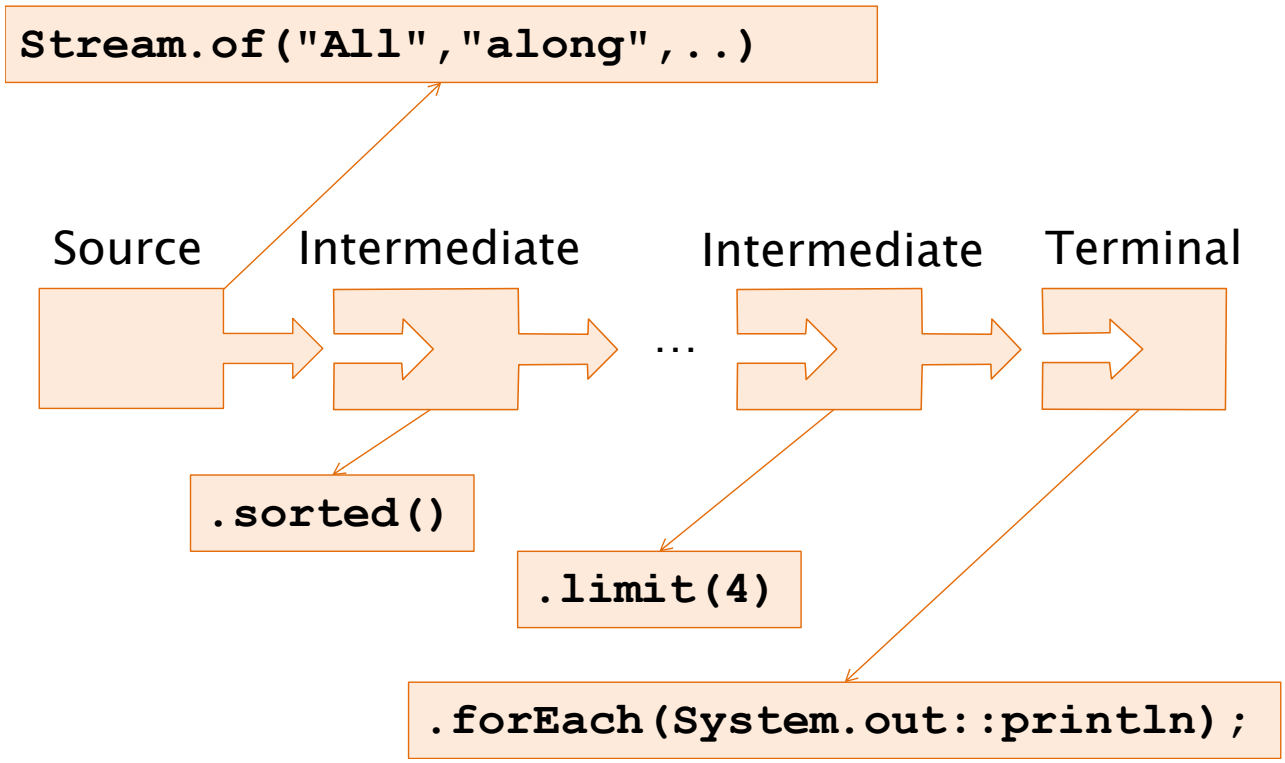


Figure 11.1.: Pipeline structure of a stream expression.

The elements flow from one stage to the next where they are processes independently.

11.1.2. External vs. Internal iteration

The operations that are parte of a Stream expression, in fact, iterate on the elements that flow throw the stream.

In general, any time an iteration is required, it can be expressed explicitly:

```
for(String word : lyrics){
    out.println(word);
}
```

or implicitly:

```
lyrics.forEach( out::println );
```

The former version, using external (or explicit) iteration

- mixes how and what,
- allows fine grained control,
- it is more verbose,
- permits multiple (nested) iterations

The latter version, using internal (implicit) iteration:

- focuses on what,
- it is more concise,
- it is less error prone,
- avoids fine grained control.

Functionally equivalent code to the one printing the first for words can be re-written using usual – explicit – iteration:

Listing 11.2 Sample lyrics print with explicit iteration

```
int count=0;
for(String word : lyrics){
    String lc = word.toLowerCase();
    System.out.println(lc);
    if(++count>=4) break;
}
```

This version, when compared to the previous Listing 11.1 version is more vebose, haderd to understand, more error prone.

Moreover, implicit iteration has the advantage of allowing transparent optimization, e.g. enabling parallel processing without changing a single line of client code.

11.1.3. Lazy evaluation

Stream pipelines are built first, without performing any processing. Then they are executed, in response to the requests of a terminal operation. In general only the minimal amount of processing is performed in order to produce the final result.

Often instead of accepting a value – whose computation might be expensive –, a `Supplier<T>` can be used, to delay the creation of objects until when and if required, e.g.: the supplier argument in `collect()` is a factory object as opposed to passing an already created accumulating object.

Lazy evaluation is achieved using a **pull architecture**. In regular (explicit code) the iteration read the elements and *pushes* them through the sequence of operations. Differently, in Stream based code, the terminal operation *pulls* the elements from the previous stage, which pulls from the previous one and so on until the source is requested the next element.

11.1.4. Kinds of Operations

Operations of a stream can be classified based on their complexity:

- **Stateless** operations: No internal persistent storage is required E.g. `map()`, `filter()`
- **Stateful** operations: Require internal persistent storage, can be
  - **Bounded**: require a fixed amount of memory, e.g. `reduce()`, `limit()`
  - **Unbounded**: require unlimited memory, e.g. `sorted()`, `collect()`

11.2. Source operations

The main Stream sources that takes the elements from an array or a list are:

Operation	Args	Purpose
`static Arrays.stream()`	`T[]`	Returns a stream from an array
`static Stream.of()`	`T...`	Creates a stream from the list of arguments or array
`default Collection.stream()`	-	Returns a stream from a collection

The static method `static Stream<T> Arrays.stream()` accepts an array and creates a stream whose elements are the elements of the array:

```
String[] s={"One", "Two", "Three"};
Arrays.stream(s).forEach(System.out::println);
```

A similar method is the `static Stream<T> Stream.of(T... values)`, that accepts a variable numbers of arguments – which can be provided as a single array – and builds a stream:

```
Stream.of("One", "Two", "Three").forEach(System.out::println);
```

A stream can be built out of any collection with the method `Stream<T> Collection.stream()`.

```
Collection<String> coll = Arrays.asList("One", "Two", "Three");
coll.stream().forEach(System.out::println);
```

Another option is to generate the elements of a stream using code that generates the element of the stream:

Operation	Args	Purpose
`generate()`	`Supplier<T> s`	Elements are generated by calling `get()` method of the supplier
`iterate()`	`T seed, UnaryOperator<T> f`	Starts with the `seed` and computes next element by applying operator to previous element
`empty()`		Returns an empty stream

Generate elements using a `Supplier`

```
Stream.generate( () -> Math.random()*10 );
```

Generate elements from a seed

```
Stream.iterate( 0, prev -> prev + 2 );
```

Warning: the two above methods generate infinite streams. It is possible to use a variation of `iterate()` that has an addition argument (*`hasNext`*), a predicate that states if a next element is available:

```
Stream.iterate( 0, n -> n<1000, prev -> prev + 2 );
```

11.3. Intermediate operations

Operation	Args	Purpose
`limit()`	`int`	Retains only first n elements
`skip()`	`int`	Discards the first n elements
`filter()`	`Predicate<T>`	Accepts element based on predicate
`sorted()`	none or `Comparator<T>`	Sorts the elements of the stream
`distinct()`	none	Discards duplicates
`map()`	`Function<T, R>`	transforms each element of the stream using the mapper function

11.3.1. Skip and Limit

It is possible to retain only the first n elements of to discard the first n elements with the methods `limit()` and `skip()`.

To use only the elements from second to fourth:

```
Stream.of(lyrics)
.limit(4)
.skip(1)
.forEach(System.out::println);
```

11.3.2. Filtering

Filtering is perfomed with method `default Stream<T> filter(Predicate<T>)` that accepts a predicate. The predicate can be a method reference returning a boolean, possibly modified with `Predicate` methods:

```
Stream.of(lyrics)
.filter(Predicate.not(String::isBlank))
.forEach(System.out::println);
```

Alternatively the predicate can be a lambda expression:

```
Stream.of(lyrics)
.filter(w -> Character.isUpperCase(w.charAt(0)))
.forEach(System.out::println);
```

Filtering is a stateless operation, each execution is independent from previous ones and requires no storage.

11.3.3. Distinct

It is possible to discard the duplicate elements in the stream with the method `default Stream<T> distinct()`.

```
Stream.of(lyrics)
.distinct()
.forEach(System.out::println);
```

The `distinct()` operation is stateless and unbounded. It has to keep track of all the elements that were already encountered to discard all the duplicates.

11.3.4. Sorted

Sorting is performed `default Stream<T> sorted()` that release the elements in ordered.

The order can be either the natural order, when providing no argument:

```
Stream.of(lyrics)
.sorted()
.forEach(System.out::println);
```

Or with specific ordering when a comparator is passed:

```
Stream.of(lyrics)
.sorted(Comparator.comparing(String::length))
.forEach(System.out::println);
```

The `sorted()` operation is stateless and unbounded. It has to buffer all the elements of the stream before being able to release the first one.

11.3.5. Mapping

The mapping operation transforms the element, the tranformation is applied with the method `default Stream<R> map(Function<T,R> mapper)`.

```
Stream.of(lyrics)
.map(String::toLowerCase)
.forEach(System.out::println);
```

11.3.6. Flat mapping

The method `<R> Stream<R> flatMap(Function<T, Stream<R>> mapper)` extracts a stream from each incoming stream element and concatenates together the resulting streams.

The context where this kind of operation might be usefule is when the stream elements are containers (e.g., `List` or array). Typically this occurs when elements are mapped to containers or arrays.

Often the next operations should be applied to all elements inside those containers, not to the containers themselves.

Typically:

- `T` is a `Collection<R>` (or a derived type) or `[]`
- mapper can be `Collection::stream` or `Stream.of()`

To print out all the distinct characters used in the text:

```
Stream.of(lyrics)
  .map(String::toCharArray)
  .flatMap(Stream::of)
  .forEach(System.out::println);
```

11.4. Terminal Operations

Operation	Return	Purpose
`findAny()`	`Optional<T>`	Returns the first element(order does not count)
`findFirst()`	`Optional<T>`	Returns the first element(order counts)
`min()`/ `max()`	`Optional<T>`	Finds the min/max element based on the comparator argument
`count()`	`long`	Returns the number of elements in the stream
`forEach()`	`void`	Applies the Consumer function to all elements in the stream
`toList()`	`List<T>`	Retuns a list containing all the elements
`toArray()`	`T[]`	Returns an array containing all the elements
`anyMatch()`	`boolean`	Checks if any element in the stream matches the predicate
`allMatch()`	`boolean`	Checks if all the elements in the stream match the predicate
`noneMatch()`	`boolean`	Checks if none element in the stream match the predicate

11.4.1. For each

All the examples above used this terminal operation that performs the given action on all elements.

A typical usage is to print all the elements:

```
Stream.of(lyrics)
.forEach(System.out::println);
```

Another option is to perform an operation on an external object:

```
StringBuffer res=new StringBuffer();
Stream.of(lyrics)
.forEach(res::append);
```

It is important to use a clase whose methods are *reentrant* (`synchronized`) so that a parallel execution of the stream does not creates race conditions leading to inconsistent results.

11.4.2. To container

The methods `toList()` and `toArray()` can be used to collect all the elements into a container or array.

11.4.3. Find

The two methods `findAny()` and `findFirst()` return the first element in the stream.

The difference is that the former allows optimization for parallel streams and therfore is not deterministic. While the latter is deterministic and returns alwasy the first element of the stream.

Both methods return an `Optional<T>` to cope with the possibility of an empty stream.

11.4.4. Min and Max

The two methods `min(Comparator<? super T> cmp)` and `max(Comparator<? super T> cmp)` return the first and last elements according to the order determined by the comparators.

```
String first =
Stream.of(lyrics)
.min(Comparator.naturalOrder())
.getOrElse("<none>");
```

Both methods return an `Optional<T>` to cope with the possibility of an empty stream.

11.4.5. Count

The method `count()` returns the number of elements present in the stream.

The return type is a `long` to minimize the risk of overflow when processing long streams.

11.4.6. Matching

The methods `boolean anyMatch(Predicate<T> predicate)`, `boolean allMatch(Predicate<T> predicate)`, `boolean noneMatch(Predicate<T> predicate)` check respectively:

- `any` if at least an element matches the predicate,
- `all` if all elements match the predicate,
- `none` if no element matches the predicate.

11.5. Numeric streams

To optimize performance when treating numeric values, and avoid the overhead induced by the wrapper classes, the library provides specialized streams for the three main numeric primitive types:

- `DoubleStream`
- `IntStream`
- `LongStream`

11.5.1. Conversion

It is possible to map to primitive streams using the conversion methods from `Stream<T>`: `mapTo`*X*`()` that are defined for the main primitive types:

- `IntStream mapToInt(ToIntFunction<T> mapper)`
- `LongStream mapToLong(ToLongFunction<T> mapper)`
- `DoubleStream mapToDouble(ToDoubleFunction<T> mapper)`

The inverse conversion can be performed using:

- `mapToObj(`*X*`Function<U> mapper)`: applies a function that converts the `int` element into any object
- `boxed()`: performs a simle boxing

11.5.2. Generators

The primitive streams provide additional generator methods, in particular both `IntStream` and `LongStream` provide a `range(start,end)` method that can be used to generate a sequence of numbers.

The class `Random` offers a few method to generate a primitive stream of random numbers:

- `DoubleStream doubles()`
- `IntStream ints()`
- `LongStream longs()`

The three methods offer four variations:

- no arguments: generates an infinite stream of random numbers
- two arguments: generates an infinite stream of random numbers in the given range
- one long argument: generates a stream of random numbers of the given length
- three argument: generates a stream of random numbers of the given length in the given range

11.5.3. Terminal operations

The numeric streams provide a set of additional terminal operations.

Operation	Return	Purpose
`sum()`	`Optional`*X*	Returns the sum of all elements
`average()`	`OptionalDouble`	Returns the average of all elements
`min()`/ `max()`	`Optional`*X*	Finds the min/max
`summaryStatistics()`	*X*`SummaryStatistics`	Returns the summary statistics of the stream elemens

11.5.4. Performance

Primitive numeric streams are more efficient since no boxing and unboxing operations are required.

For instance to find the maximum value of a stra Numeric streams

```
Random rnd = new Random(1971);
int[] ints = rnd.ints(N, 0, 10000).toArray();
int max = IntStream.of(ints).max().getAsInt();
```

```
List<Integer> lints = IntStream.of(ints).mapToObj(Integer::valueOf).toList();
int max = lints.stream().max(Comparator.<Integer>naturalOrder()).get();
```

Stream type	Performance
`IntStream`	0.6ns per element
`Stream<Integer>`	2.6ns per element

The direct comparison of `int` values is approximatively four times more efficient that the usage of the `naturalOrder()` comparator.

11.6. Reducing

The reduce operation combines the elements of the stream together to produce a final result. It can be performed through:

- `reduce(BinaryOperator<T> reducer)`: produces the temporary result by combining the previous result with the next element with the `reducer`;
- `reduce(T identity, BinaryOperator<T> reducer)`: starts with a result that is the `identity` then computes the next result by combining the previous result with the next element;
- `reduce(U identity, BiFunction<U,T,U> reducer, BinaryOperator<U> combiner)`: same as above but the result is not the same type as the elements, a `combiner` is used to combine results from parallel streams.

Reduces the elements of this stream, using the provided identity value and an associative merge function

```
int maxLen =
Stream.of(lyrics)
.distinct()
.map(String::length)
.reduce(0, Math::max);
```

The above reduce computes the max as described in Figure 11.2.

Reduce operation can be easily parallelized since the operation is the same for any pair of elements.

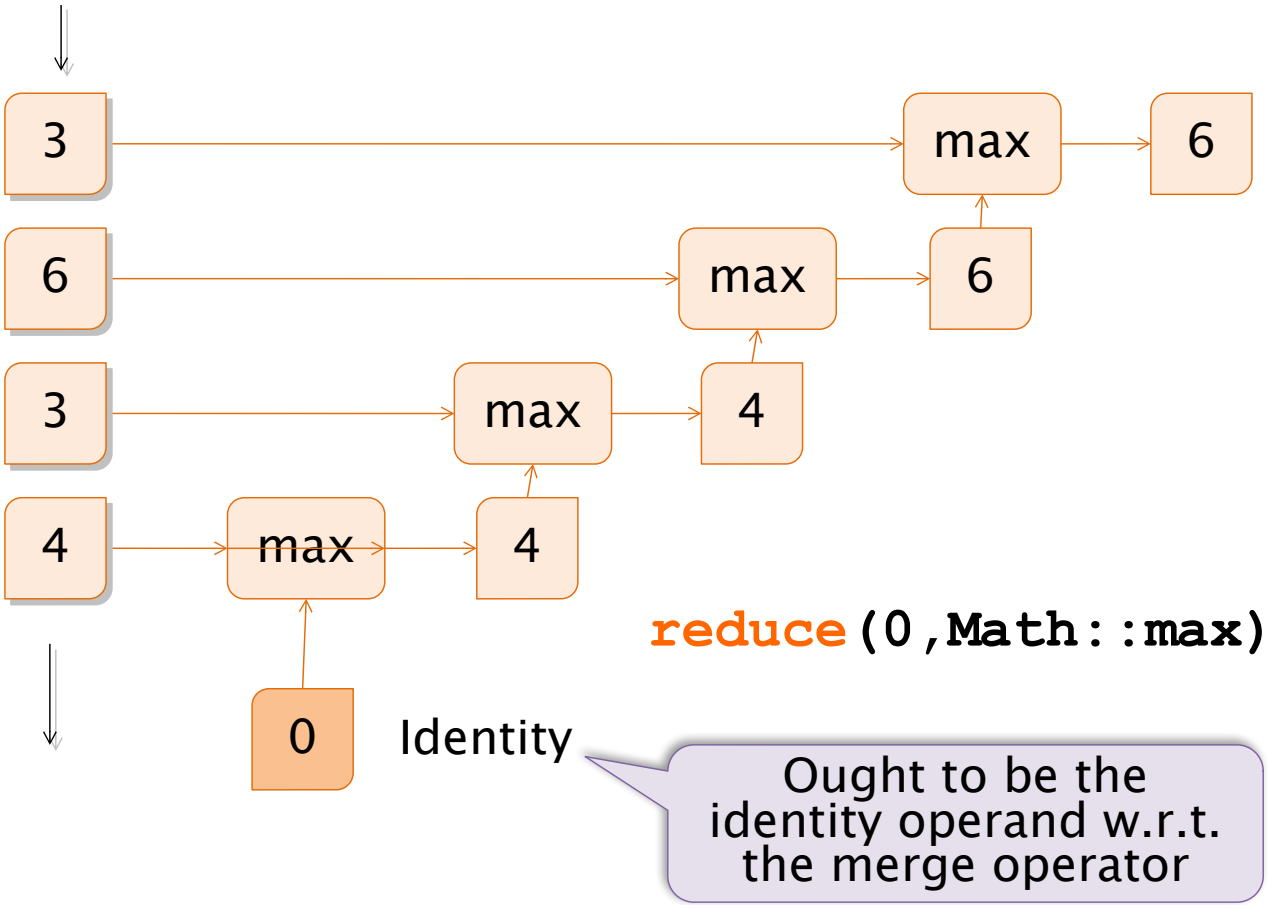


Figure 11.2.: Exampe of reduce operation

```
IntStream.of(numbers).reduce(0,Math::max);                    ①
IntStream.of(numbers).parallel().reduce(0,Math::max);         ②
```

① serial reduce
② parallel reduce

The resulting performance of the two alternatives is:

Stream kind	Performance
Serial	0.6ns per item
Parallel	0.1ns per item

Potentially up to n times faster, where n is the number of CPU cores available. In practice synchronization issues can introduce a significant overhead.

11.7. Collecting

The methdo `Stream.collect()` takes as argument a recipe for accumulating the elements of a stream into a result container.

The recipe defined by the collector includes three elements:

- a *supplier* providing a result container object
- an *accumulator* that accumulates a new element into the result
- a *combiner* that merge two intermediate results (typically used in case of parallel streams)

To accumulate the elements into a list:

```
List<Integer> n = Stream.of(numbers).
collect(LinkedList::new,                                      ①
        List::add,                                            ②
        List::addAll);                                        ③
```

① supplier of the container result
② accumulator
③ combiner

The operations performed by the collecter defined as above is described in Figure 11.3.

Note that the above operation should be avoided, the method `toList()` of `Stream` is a much more compact and efficient alternative.

Collection is a stateful operation and unbounded operation. The memory occupation depends on the number of elements processed.

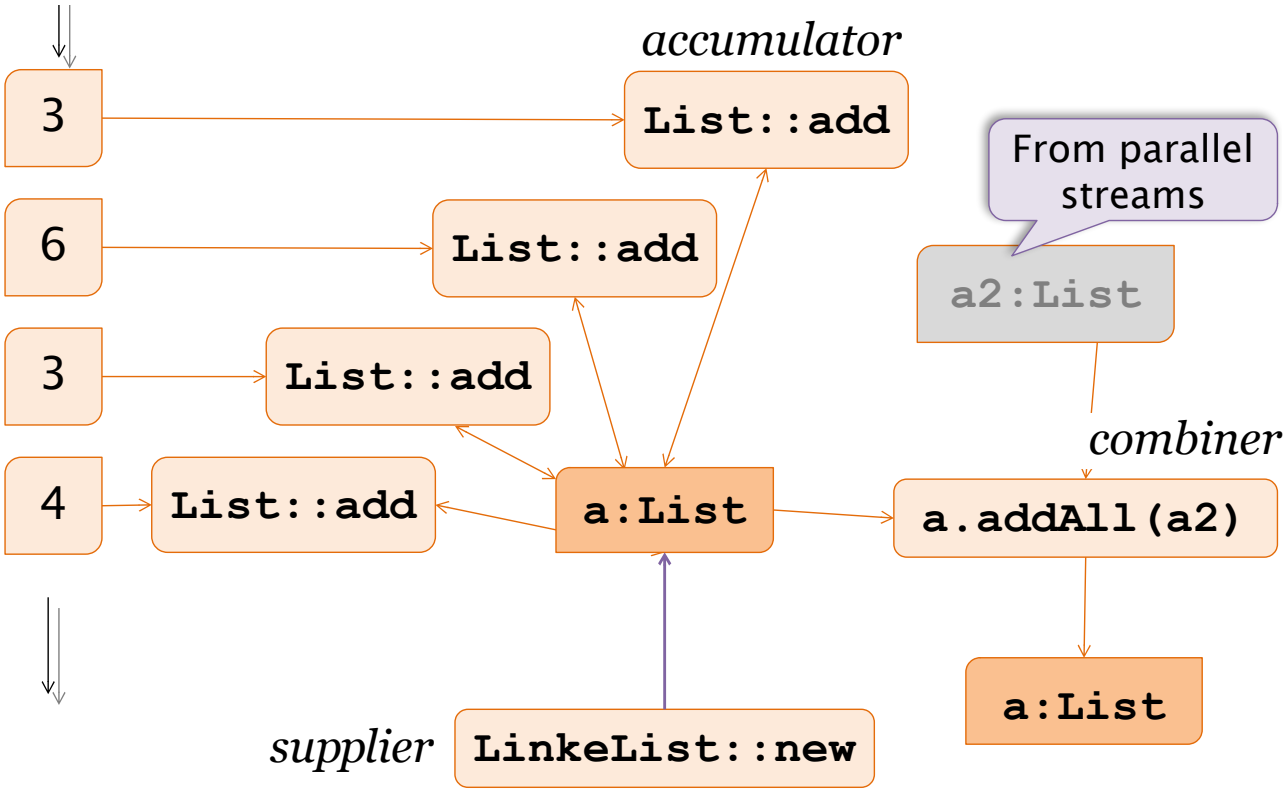

accumulator
3
List::add
From parallel streams
6
List::add
a2:List
3
List::add
combiner
4
List::add
a:List
a.addAll(a2)
a:List
supplier
LinkeList::new

Figure 11.3.

11.7.1. Collect vs. Reduce

The main difference between the collect and reduce operations are:

- Reduce is bounded, the merge operation can be used to combine results from parallel computation threads
- Collect is unbounded, combining results form parallel computation threads can be performed with the combiner

11.7.2. Predefined Collectors

Java Stream API provides a number of predefined collectors. Predefined recipes are returned by static methods in `Collectors` class

The methods are easier to access through:

```
import static java.util.stream.Collectors.*;
```

For insatance to compute the average of a sequence of numbers

```
double averageWord = Stream.of(txta)
    .collect(averagingInt(String::length));
```

A few collectors summarize the sequence of elements in the stream. Such elements.

Collector	Return	Purpose
`counting()`	`long`	Count number of elements in stream
`maxBy()` / `minBy()`	T (elements type)	Find the min/max according to given Comparator
`summing`*Type*`()`	*Type*	Sum the elements
`averaging`*Type*`()`	*Type*	Compute arithmetic mean
`summarizing`*Type*`()`	*Type*`Summary` Statistics	Compute several summary statistics from elements

Accumulating collectors accumulate the elements into group containers.

Collector	Return	Purpose
`toList()`	`List<T>`	Accumulates into a new List
`toSet()`	`Set<T>`	Accumulates into a new Set (i.e. discarding duplicates)
`toCollection(Supplier<>` `cs)`	`Collection<T>`	Accumulate into the collection provided by given `Supplier`
`joining()`	`String`	Concatenates into a String Optional arguments: separator, prefix, and postfix

To find out the three longest words in text:

```
List<String> longestWords = Stream.of(txta)
.filter( w -> w.length()>10)
.distinct()
.sorted(comparing(String::length).reversed())
.limit(3)
.collect(toList());
```

What if two words share the 3rd position?

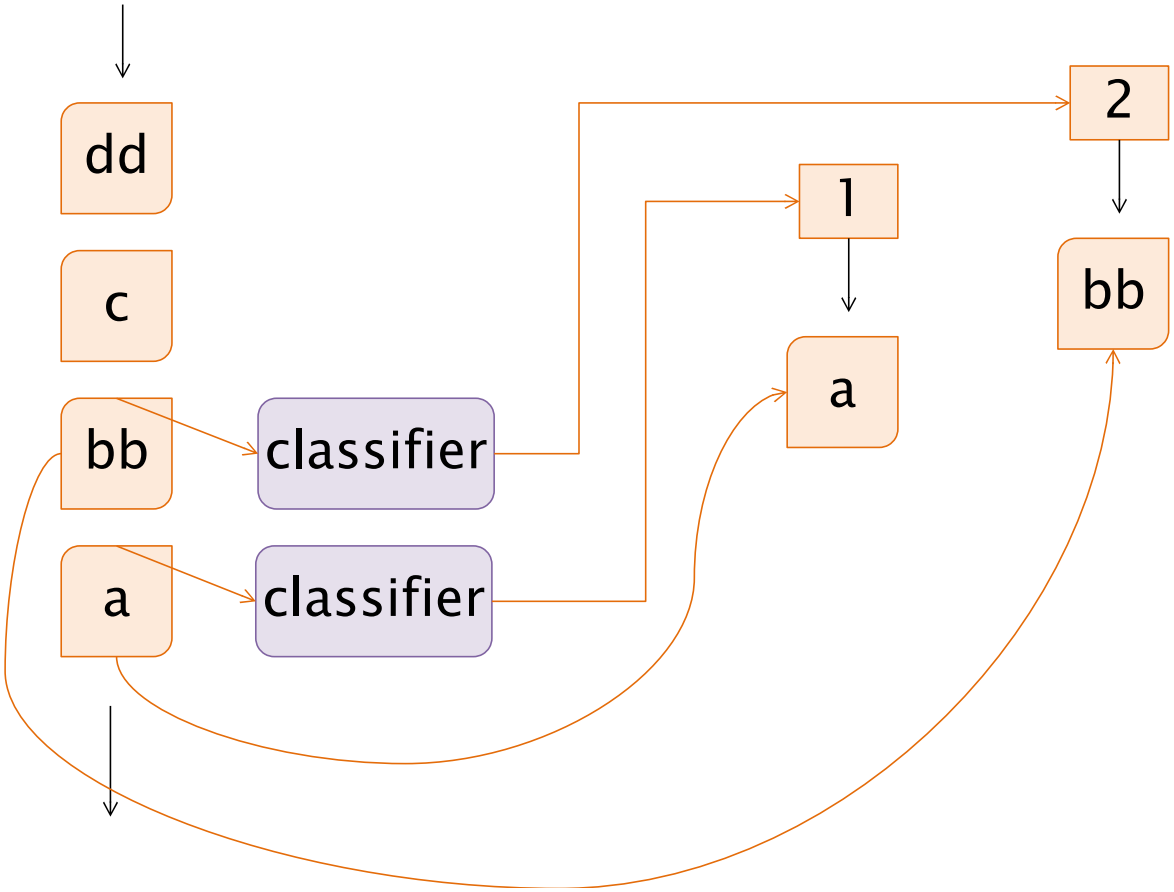
Grouping collectors divide the elements into separate streams, by means of a classifier function (or predicate). Each separate stream is then further collected – by default into a list – and placed into a map with the corresponding classification label as the key.

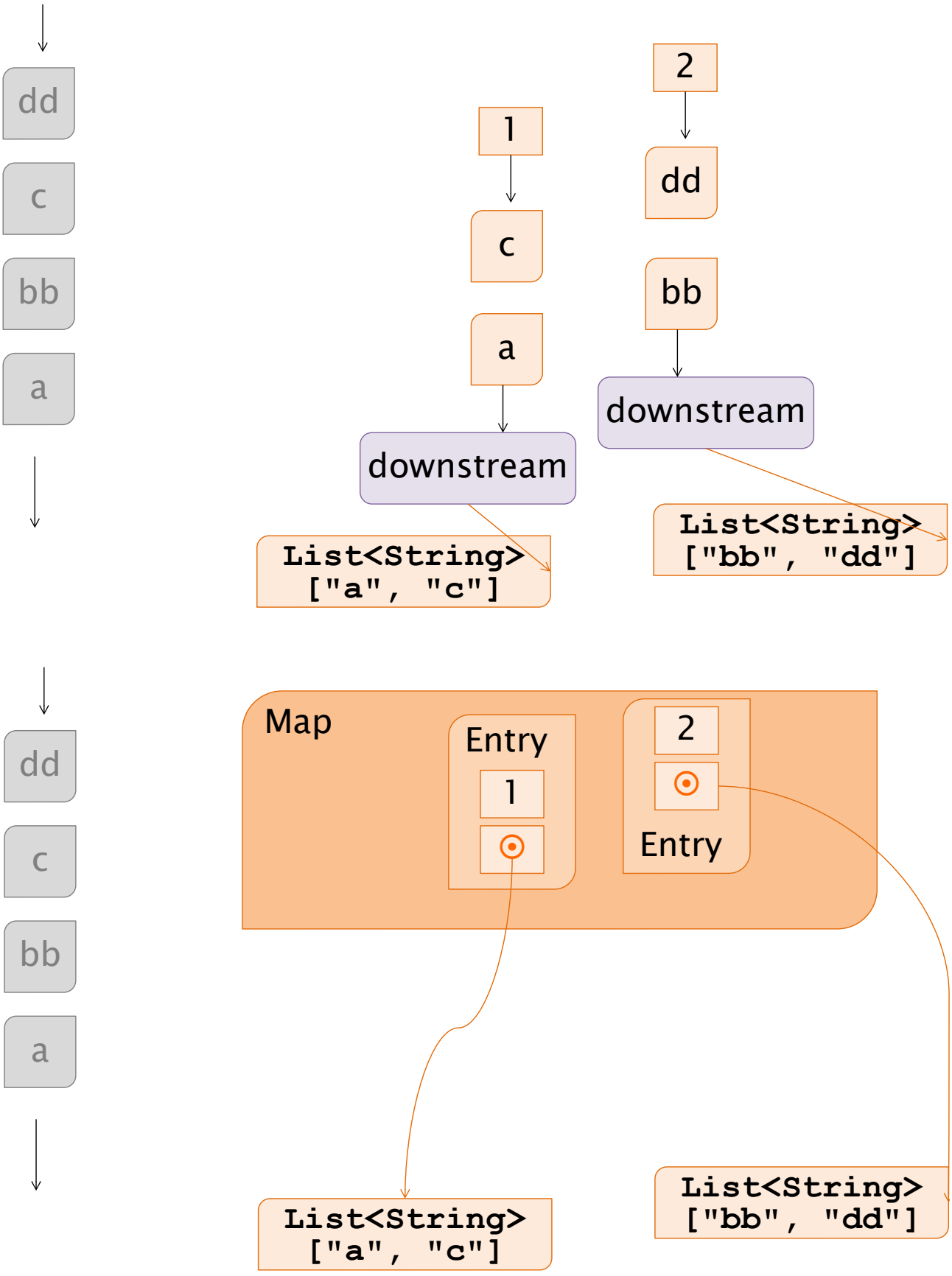
Collector	Return	Purpose
`groupingBy(Function<T,K> classifier)`	`Map<K, List<T>>`	Maps according to the key extracted (by classifier) and adds to list.
`partitioningBy(Predicate<T> p)`	`Map<Boolean, List<T>>`	Split according to partition function (p) and add to list.

The method `groupingBy()` uses the classifier argument to define the keys of the resulting map and to separate the elements into separate streams that are then collected into lists.

For instance to group the words into group according to their length:

```
Map<Integer,List<String>> byLength =
    Stream.of(lyrics)
    .distinct()
    .collect(groupingBy(String::length));
```


dd
c
bb
a
classifier
classifier
1
a
2
bb


dd
c
bb
a
1
c
a
downstream
List<String>
["a", "c"]
2
dd
bb
downstream
List<String>
["bb", "dd"]
dd
c
bb
a
Map
Entry
1
2
Entry
List<String>
["a", "c"]
List<String>
["bb", "dd"]

The method accepts two additional optional arguments, a map factory supplier and the downstream collector. The above code, using the default behavior, can be rewritten as:

```
Map<Integer,List<String>> byLength1 =
    Stream.of(lyrics)
    .distinct()
    .collect(groupingBy(
        String::length,                                    ①
        HashMap::new,                                      ②
        toList()                                           ③
    ));
```

① classifier, determines the key and the stream split criterion
② map factory, build the result container
③ downstream collector, collects the spearate stream

The map factory can be leveraged to create, e.a. a sorted map

```
Map<Integer,List<String>> byLength =
    Stream.of(lyrics).distinct()
    .collect(groupingBy(String::length,
                ()-> new TreeMap<>(reverseOrder()),        ①
                toList()))
```

① Map sorted by descending length

The downstream collector can be used to collect into a something different than a `List`.

```
Map<String,Long> frequency =
Stream.of(lyrics)
.collect(groupingBy(
                Function.identity(),                       ①
                counting()));                              ②
```

① Grouped by word, i.e. the very same element
② Downstream is `counting`

To create the ranking of words by frequency we can take the result of the frequency – as computed above –, sort the entry set by the key – the value – and map the sorted entries to strings:

```
List<String> freqSorted =
Stream.of(lyrics)
.collect(groupingBy(Function.identity(), counting()))
.entrySet().stream()
.sorted(
  comparing(Map.Entry<String,Long>::getValue)
  .reversed())
.map( e -> e.getValue() + ":" + e.getKey())
.collect(toList());
```

11.7.3. Collector Composition

Collector	Purpose
`collectingAndThen(Collector<T,?,R> cltr, Function<R,RR> mapper)`	Applies a transformation (mapper) after performing collection (cltr)
`mapping(Function<T,U> mapper, Collector<U,?,R> cltr)`	Performs a transformation (mapper) before applying the collector (cltr)

The ranking computed above could be also expressed using the `collectingAndThen()` composition method:

```
Stream.of(lyrics)
.collect(collectingAndThen(
    groupingBy(Function.identity(), counting())                          ①
,
    m->m.entrySet().stream()                                             ②
    .sorted(comparing(Map.Entry<String,Long>::getValue).reversed())
    .map(e->e.getValue()+ ":" +e.getKey())
    .collect(toList())
));
```

① collecting
② and then

Note the type parameter specification `<String,Long>` that is required to let the compiler correctly infer the type of the comparator.

An alternative way of computing the ranking and storing it into a sorted map is:

```
Map<Long,List<String>> ranking =
Stream.of(lyrics)
.collect(collectingAndThen(
    groupingBy( Function.identity(), counting())                         ①
,
    m->m.entrySet().stream()                                             ②
    .collect(groupingBy(
        Map.Entry::getValue,
        ()->new TreeMap<>(reverseOrder()),
        mapping(Map.Entry::getKey,
                toList())
    ))
));
```

① collecting
② and then

11.7.4. Custom collectors

The interface `Collector` is defined as follows:

```
interface Collector<T,A,R>{
Supplier<A> supplier()                                    ①
BiConsumer<A,T> accumulator();                            ②
BinaryOperator<A> combiner();                             ③
Function<A,R> finisher();                                 ④
Set<Characteristics> characteristics();                   ⑤
}
```

① Creates the accumulator container
② Adds a new element into the container
③ Combines two containers (used for parallelizing)
④ Performs a final transformation step
⑤ Capabilities of this collector

In addition to implementing the interface, it is possible to build a collector using the builder function `Collector.of()`:

```
static Collector<T,A,R> of(
    Supplier<A> supplier,
    BiConsumer<A,T> accumulator,
    BinaryOperator<A> combiner,
    Function<A,R> finisher,
    Characteristic... characteristics)
```

Using this function, possibly with method references and/or lambda functions is more compact than extending the interface Collector

```
Collector<String,List<String>,List<String>> toList =
Collector.of(
    ArrayList::new,                                       ①
    List::add,                                            ②
    (a,b)->{a.addAll(b);return a;},                       ③
    Collections::unmodifiableList                         ④
);
```

① supplier
② accumulator
③ combiner
④ finisher

The `Characteristics` class defines a set of constants that specify the features of the collector:

- `IDENTITY_FINISH` Finisher function is the identity function therefore it can be elided
- `CONCURRENT` Accumulator function can be called concurrently on the same container

- `UNORDERED` The operation does require stream elements order to be preserved

Such `Characteristics` can be used to optimize execution. If both CONCURRENT and UNORDERED, then, when operating in parallel, the accumulator method is invoked concurrently by several threads and the combiner is not used.

An example of a collector used to compute the average length of a stream of String, eses the AverageAcc accumulator object:

```
Collector<Integer,AverageAcc,Double>
avgCollector =  Collector.of(
            AverageAcc::new,    // supplier
            AverageAcc::addWord,// accumulator
            AverageAcc::merge , // combiner
            AverageAcc::average // finisher
);
```

The class `AverageAcc` can be defined ase:

```
class AverageAcc {
    private long length;
    private long count;
    public void addWord(String w){
     this.length+=w.length();// accumulator
        count++;    }
    public double average(){   // finisher
        return length*1.0/count;    }
    public AverageAcc merge(AverageAcc o){
        this.length+=other.length;
        this.count+=other.count;  // combiner
        return this;}
}
```

11.8. Exceptions and Functional Interfaces

Functional interfaces largely used in Stream API do not include exceptions. Lambdas and method reference are not allowed to throw (checked) exceptions This is mainly due to the intermediate code that does not handle exceptions. One motivation for such a choice is the additional complexity that would arise when type parameters also include exceptions.

Unchecked exceptions works as usual and interrupt the stream processing

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> Integer.parseInt(s) )                                    ①
.sum();
```

① a `NumberFormatException` interrupts all the processing.

In case code throwing unchecked exception needs to be used within stream operations, a few possible strategies can be applied:

- Bury the exception: catch the exception and ignore it
- Wrap the exception: use `RuntimeException` pointing to the original one
- Sneaky exceptions: throw as unchecked (using a trick)
- Annotate the results: insert additional information in the result

11.8.1. Bury the exception

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> {
    try{ return convert(s); }
    catch(NotANumber e)
    { return 0; }
})
.sum();
```

11.8.2. Wrap the exception

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> {
    try{ return convert(s); }
    catch(NotANumber e)
    {throw new RuntimeException(e);     }})
.sum();
```

11.8.3. Sneakily throw the exception

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> {
    try{ return convert(s); }
    catch(NotANumber e)
    {sneakyThrow(e);
     return -1; }})
.sum();
```java

Sneaky throw method: using generics type erasure it is possible to "uncheck" an exception

```java
static <E extends Throwable>
```

```
void sneakyThrow(
    Exception exception) throws E {     throw (E)exception;
}
```

11.8.4. Annotate the results:

A suitable class must be defined that hosts:

- the result of the computation
- an "annotation" about anomalies, typically an exception

The operations are perfomed on the results and the possible exception annotation is carried on.

```
try{
    Annotated<Integer,ConversionException> result =
    Stream.of("1","2","B","6","30")
        .map( Annotated.applyWithAnnotation(StreamExceptions::convert) )
        .reduce( Annotated.identity(), Annotated.reduceWith(Integer::sum) );

    IO.println("Result: " + result.get());
}catch(ConversionException ce){
    IO.println("Trouble in computing: " + ce.getMessage());
}
```

An example of annotated result class is the following one:

```
public class Annotated<T,E extends Exception>{
    final T item;
    final E note;
    private Annotated(T i, E e){
        this.item=i; this.note=e;
    }
    public T getOrElse(Function<E,T> f){
        if(note==null) return item;
        return f.apply(note);
    }
    public T get() throws E {
        if( note!=null ) throw note;
        return item;
    }
    public boolean isClean() {
        return note==null;
    }
    public E annotation(){
        return note;
    }
    static <T, E extends Exception>
```

```
    BinaryOperator<Annotated<T,E>> reduceWith(BinaryOperator<T> op){
        return (a,b) -> {
            if(a.item==null) return b;
            return
            new Annotated<>(b.item==null?a.item:op.apply(a.item,b.item),
                            a.annotation==null?b.annotation:a.annotation);
        };
    }
    public static <T,E extends Exception>
    Annotated<T,E> identity(){
            return new Annotated<>(null,null);
    }

    public static <T,U,E extends Exception>
    Function<T,Annotated<U,E>> annotate(ThrowingFunction<T,U,E> f){
        return x -> {
            try {
                return new Annotated<>(f.apply(x),null);
            } catch (Exception e) {
                return new Annotated<>(null, (E)e);
            }
        };
    }

    interface ThrowingFunction<T,U,E extends Exception> {
        U apply(T x) throws E;
    }
}
```

11.9. Summary

- Streams provide a powerful mechanism to express computations of sequences of elements
- The operations are optimized and can be parallelized
- Operations are expressed using a functional notation
- More compact and readable w.r.t. imperative notation

12. JUnit

JUnit is a testing framework for Java programs. It was originally written by Kent Beck and Erich Gamma. It is a framework with unit-testing functionalities It is integrate as a plug-in in most IDEs, making it seamless to execute the test during the routing code development activities.

The official site is: http://www.junit.org

Unit tests are intended to verify the correctness of units of a program. During testing, units are considered in isolation. In Java, units are typically classes and packages.

Unit testing is particularly important when software requirements change frequently. Code often need be refactored to incorporate the changes. Unit tests help ensure that the refactored code continues to work.

JUnit Framework JUnit helps the programmer: Define and execute tests and test suites Formalize requirements and clarify architecture Write and debug code Integrate code and always be ready to release a working version

12.1. History

The key steps in the history of JUnit are:

- 1997 on the plane to OOPSLA97 Kent Beck and Erich Gamma wrote JUnit
- Junit.org – August 2000
- Junit 3.8.1 – September 2002
- Junit 4.0 – February 2006, the latest release: 4.13.2 - Feb 2021
- Junit 5.0 – September 2017, the latest release: 5.12.0 – Feb 2025

12.2. JUnit at work

Each test case in JUnit is represented as a test method. A test method returns no result, it just performs operations on the code under test and checks the results. Checks are performed using a set of `assert*()` methods. The JUnit framework detects the anomalies and reports them.

The basic testing procedure that JUnit adopts is

- For each test (method) Junit:
  - calls pre-test fixture Intended to acquire resources and create any objects that may be needed for testing
  - calls the test method The actual test that checks the output of the element under test

  - calls post-test fixtures Intended to release resources and remove any objects created that is no longer needed

12.2.1. Assertions

The main standard *`assert*()`* methods are;

- `assertTrue(boolean test)`
- `assertFalse(boolean test)`
- `assertEquals(expected, actual)`
- `assertSame(Object exp, Object actual)`
- `assertNotSame(Object exp, Object act)`
- `assertNull(Object object)`
- `assertNotNull(Object object)`
- `fail()`

We can test a boolean condition using `assertTrue(condition)`. If the tested condition is

- true => it proceeds with execution
- false => aborts the test method execution, and prints out the optional message

We can test the opposite condition using the `assertFalse(condition)` that fails if condition is true.

We can test the equality of primitives and objects using the `assertEquals( expected, actual)` method. For floating point values we need to express a threshold `assertEquals(expected,actual,err)`, this is due to the fact that floating point operation might produce approximantion errors, therefor a precise comparison in this case might produce the wrong result. Ex. `assertEquals(1.0, Math.cos(3.14), 0.01);`

All *`assert*`* method have an overload version that include also a message that is reported when the assertion failes.The method is in fact optional, but it is recommended to provided it, so that debugging a failing test is easier.

All JUnit versions up to 4 use the following method signature:

```
static void assertTrue( String message, boolean test)
```

While JUnit 5 adopted a slighly different signature with the message as the last parameter. In addition it is possible to provide a supplied instead of a precomputed message:

```
static void assertTrue( boolean test, String message )
static void assertTrue( boolean test, Supplier<String> messageSupplier)
```

12.2.2. Fixtures

The *fixtures* are portions of test cases that are cloned in several different tests. They are typically used for initializing or releasing resources.

There are two main types of fixtures:

- Pre-test (set up)
- Post-test (tear down)

JUnit allaows placing the fixtures in separate methods to avoid duplication and shorten test cases. Such fixtures are automatically executed before and after tests. Therefore they are written once and avoid the risk of forgetting some initialization.

12.2.3. Failures vs. Errors

The outcoma of a test can be

- **Success**: all the code in the method has been executed and all assertions have been verified, in this case we say that the test is *passed*.
- **Failure**: an `assert*()` method found the condition it checked is not verified, in practice the code under test produced an output, but it is not the expected one. During the execution of the tests an error was found (e.g., NullPointerException)
- **Error**: an unexpected exception is detected during the execution of the test method, the program could not produce any output due to an error.
- **Skipped**: the test has not been executed because some pre-condition was not met.

12.2.4. Skipping tests

There may be several reasons to skip a test:

1. I wrote a test for a specific requirement, but that requirement hasn't been implemented yet.
   It is pointless to see a failure or an error for such a test when I *already know perfectly well* that I haven't done that part yet.
2. Test execution requires a lot of time for large applications, so to speed up the excution I could skip some tests that are less critical and concentrate the attention on other more important tests.
3. There could be conditions not met by the system where tests are executed.

   - Imagine you have an app that should perform some action **using a fingerprint reader**, but I'm running the test on a computer **without** any fingerprint reader; I know for sure that it would fail because the hardware is missing, it is better to skip the test because a failure would tell me nothing about the code.
   - I might have a test specific for MacOS, another one specific for Windows, and another specific for Linux; obviously, depending on the operating system I'm running the test on, some tests will run, others won't.

12.2.5. Testing exceptions

In presence of methods throwing exceptions, at least two main cases shall be checked:

- a normal behavior is expected, therefore no exception should be thrown, in this case if that exception is raised an error is reported.
- an anomaly is expected, therefore an exception should be thrown, i n this case the tests fails if NO exception is detected.

12.2.6. Sample class: Stack

```
public class Stack {
    private int[] stack;
    private int next = 0;

    public Stack(){ this(10);}
    public Stack(int size){ stack = new int[size];}

    public boolean isEmpty(){ return next==0; }

    public boolean push(int i) {
        if(next==stack.length) return false;
        stack[next++] = i;
        return true;
    }
    public int pop() throws StackException {
        if(next==0) throw new StackException()
        return stack[--next];
    }
}
```

12.3. Junit 4 Syntax

JUnit 4 – differently from previous versions – introduced the use of java annotations to make different portion of the tests.

The adaventages are:

- Less constraints on names
- Easier to read/write
- Backward compatible with JUnit 3

12.3.1. Assertions

assert*() methods assertThat() method To use the Hamcrest matchers

12.3.2. Test a Stack

```
public class TestStack {
@Test
public void testStack() throws StackException {
    Stack aStack = new Stack();
    assertTrue("Stack should be empty",
                      aStack.isEmpty());
    aStack.push(10);
    assertFalse("Stack should not be empty!",
                       aStack.isEmpty());
    aStack.push(-4);
    assertEquals(-4, aStack.pop());
    assertEquals(10, aStack.pop());
 }
}
```

12.3.3. Running a test case

The JUnit runner executes all methods that satisfy the following conditions:

- are annotated with "`@Test`"
- have `public` visibility
- return `void`
- accept no arguments `()`

The runner ignores the rest of the class. It is common practice to include in the class attributes and helper methods provided they are not annotated or not public.

12.3.4. Fixtures

Annotate a method with `@Before` to make it a pre-test fixture:

- It is executed before each test method is run
- It is intended to initialize the objects that will be used by test methods
- There is no limit to the setup you can do in a pre-test method
- Helps reducing duplication of code

Annotate a method with `@After` to make it a post-test fixture:

- It is executed after each test method is run
- It is intended to release system resources (such as streams)
- In most cases, a post-test fixture is not required
- Before the next test is executed the pre-test fixture is run again so attribute will be re-initialized

12.3.5. Testing Exceptions

There are two alternative approaches to testing methods in a settinge where they are expected to throw an exception:

- declare it in the annotation,
- use a specific assert statement.

When an exception is expected from the test it can be declared with the annotation

```
@Test(expected=StackException.class)
public void testEmptyPop() throws StackException {
  Stack aStack = new Stack();
  aStack.pop();
}
```

If not declared any exception thrown will be counted as an Error

Alternatively, it is possivle to assert that an exception is expected:

```
assertThrows("message",
             Exception.class,
             ()-> { /* exception throwing code */ })
```

In the case of the `Stack` class to test that a *pop* operation on an empty stack produces an exception we can use the following code:

```
@Test
public void testEmptyErrorAssert() {
  Stack aStack = new Stack();
  assertThrows("Expected excecption on empty pop",
               StackException.class,
               ()-> aStack.pop());
}
```

When an exception is not expected, it must be declared, otherwise we have an unhandled exception error:

```
@Test
public void testPop() throws StackException {
  Stack aStack = new Stack();
  aStack.push(1);
  aStack.pop();
}
```

12.3.6. TestSuite

Allows running a group of related tests as a single batch:

```
@RunWith(Suite.class)
@SuiteClasses({
    TestStack.class, AnotherTest.class
})
public class AllTests { }
```

12.3.7. Skipping tests

Some tests can be intended to be skipped, e.g., not ready yet:

```
@Ignore("Incomplete test")
```

Other tests can be executed only is some condition are met, e.g., a resource is available

```
assumeTrue("X not available", res.isAvailable())
```

12.3.8. Summary of JUnit 4 Annotations

- `@Test`: marks test methods
- `@Before` and `@After` Mark pre and post fixtures
- `@Ignore` Mark method or class to be skipped
- `@RunWith(Suite.class)` declare a suite manager class
- `@Suite.SuiteClasses({ … })` define the classes of the suite

All classes and annotations in JUnit 4 are defined in package ‘org.junit.

Assertions are made available with

```
import static org.junit.Assert.*;
import static org.junit.Assume.*;
```

Annotations must be imported as

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

Suites require:

```
import org.junit.runners.Suite;
import org.junit.runners.Suite. SuiteClasses;
```

12.4. Junit 5

Uses Java annotations but different from 4:

- `@Before`/`@After` -> `@BeforeEach`/`@AfterEach` `@BeforeClass`/`@AfterClass` -> `@BeforeAll`/`@AfterAll`

Test methods not necessarily public

Java8 Lambda support, extensions, parameterized tests, etc.

Suites:

```
@RunWith(JUnitPlatform.class)
@SelectClasses({ … })
@SelectPackages({ … })
```

12.4.1. Test a Stack

```
public class TestStack {
    @Test
    public void testStack() throws StackException{
    Stack aStack = new Stack();
    assertTrue(aStack.isEmpty(),
                    "Stack should be empty");
    aStack.push(10);
    assertFalse(aStack.isEmpty(),
                    "Stack should not be empty!");
    aStack.push(-4);
    assertEquals(-4, aStack.pop());
    assertEquals(10, aStack.pop());
    }
}
```

Expected exception test

```
@Test
public void testSomething(){
  // e.g. method invoked with "wrong" args
   assertThrows(PossibleException.class,()->{
    obj.method("Wrong Argument")
 });
}
```

12.4.2. Skipping tests

Some tests can be intended to be skipped, e.g., not ready yet:

```
@Disabled("Incomplete test")
```

Other tests can be executed only is some condition are met, e.g., a resource is available

```
assumeTrue("X not available", res.isAvailable())
```

12.4.3. TestSuite

Indicate the JUnitPlatform runner Select classes with SelectClasses

```
@RunWith(JUnitPlatform.class)
@SelectClasses({
    TestStack.class, AnotherTest.class
})
public class AllTests { }
```

12.4.4. Summary of JUnit 5

All classes and annotations are in package `org.junit.jupiter.api`

Assertions are made available with

```
import static org.junit.jupiter.api.Assertions.*;
```

Annotations have to be imported as

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

Suites require:

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
```

12.5. Using JUnit

12.5.1. Test-Driven Development

Specify part of a feature yet to be coded Run the test and see it fail (red bar) Write code until the test pass (green bar) Repeat until whole feature implemented Refactor while maintaining the bar green

Keep your code clean: keep the bar green

12.5.2. Regression Testing

12.5.3. Bug Reproduction

When a bug is reported, specify the expected correct outcome.

See the test fail, i.e., reproduce the bug

Modify the code and adjust it until the bug-reproducing tests pass.

Check for regressions, with the existing test suites

12.5.4. Guidelines

Test should be written before code

Test everything that can break

Run tests as often as possible

> Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test

M.Fowler

12.5.5. Limitations of unit testing

JUnit is designed to call methods and compare the results they return against expected results

This ignores:

- Programs that do work in response to GUI commands
- Methods that are used primary to produce output

Heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects. This can actually be a good thing!

Methods that just return results, without side effects (such as printing), are simpler, more general, and easier to reuse

12.6. References

- K.Beck, E.Gamma. Test Infected: Programmers Love Writing Tests http://members.pingnet.ch/gamma/junit.htm
- Junit 5 home page https://junit.org
- Junit 4 home page https://junit.org/junit4/
- Hamcrest matchers http://hamcrest.org/JavaHamcrest/
- AssertJ – Fluent assertions http://joel-costigliola.github.io/assertj/index.html

13. Date and Time

In Java, there are many classes and interfaces used to represent date and time. These have evolved significantly, starting from relatively simple representations to much more sophisticated tools for properly handling calendars, timezones, and time.

Keep in mind that handling date and time is inherently complex.

For example, if in a meeting taking place in Turin someone tells you it's 5:17 PM, you understand it because you are all in Italy and you know what that means *locally*. But the same statement, if streamed to someone in the U.S., would be misleading — they're in a different time zone - so it wouldn't be 5:17 PM for them. Time zone management is critical in date and calendar handling.

Even something as seemingly trivial as computing "the next day" isn't as simple as adding one to the day digits. Months have different lengths, and leap years add complexity - like February sometimes having 29 days every fourth year -. Going even further, every so often a leap second is added to adjust for the Earth's rotational slowdown.

Another importan aspect is daylight saving time. When converting between time zones, it's not enough to know the offsets, you need to know when daylight saving rules apply. Java handles this using built-in zone databases.

If you ignore these details, you risk inaccurate results. That's why a simplified or naïve approach to calendar management often doesn't work except for the most basic operations.

Chronologically date and time in Java evolved through several different APIs:

- Timestamps (in `java.lang.System`)
- `java.util.Date`
- `java.util.Calendar`
- `java.time` package

13.1. System timestamps

The `System` class provides two methods that can be used to retrieve system timestamps:

- `currentTimeMillis()` the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC
- `nanoTime()` current value of the running JVM's high-resolution time source, in nanosecond. Does not have a fixed reference point and is often reset based on internal CPU timing. It's useful for measuring elapsed time, but not for real-world timestamps.

13.1.1. Performance measurement

A typical application of these method is to measure the time performance of several algorithms, e.g.

```
long t0 = System.nanoTime();
algorithm();
long t1 = System.nanoTime();
IO.println("Elapsed: " + (t1-t0));
```

13.1.2. Monitoring

Another typical usage is the monitoring of performance of long running operations.

A monitoring listerner interface can be used to receive apdate about the progress:

```
interface ProgressListener {
    void progress(int current, int total);
}
```

An example of a long running algorithm (concatenating integer into a string in the most inefficient way):

```
static void monitoredAlgorithm(ProgressListener l){
    String s = "";
    for(int i=0; i<N; ++i){
        if(i%1000 == 0) l.progress(i, N);                    ①
        s+=i;
    }
    l.progress(N, N);
}
```

① every given number of iterations (1000 in this case) the listener method is invoked to keep track of the progress.

A possible example of monitor:

```
long[] time = {-1,-1};
monitoredAlgorithm( (c,t) -> {
    double prop = c / (double)t * 100;                       ①
    if(time[0] == -1){                                       ②
        time[0] = System.currentTimeMillis();
        IO.print(" %.2f%% completed\r".formatted(prop));
    }else{
        time[1] = System.currentTimeMillis();                ③
        double elapsed = (time[1] - time[0])/1000.0;         ④
        double throughput = c / elapsed;                     ⑤
        double residual = t / throughput - elapsed;          ⑥
```

```
        IO.print(" %.1f%% completed in %.1f s (%.1f s remaining) throughput: %.1f k per sec\r"
                .formatted(prop, elapsed, residual, throughput/1000));
    }
});
IO.println("\nDone.");
```

① a proportion of completed work is computed
② on first invocation, the initial time is recorded and the proportion printed
③ on following invocations, the current last time is recorded
④ the elapsed time is computed
⑤ the average throughput is computed
⑥ an estimated residual time is computed
⑦ all the above information is printed

13.2. Old APIs

13.2.1. Date

In earlier versions of Java, the `Date` class in package `java.util` was introduced. It's essentially a wrapper around a `long` timestamp (milliseconds since the epoch).

However, `Date` had serious limitations, especially for time zone conversions and date arithmetic.

Even its constructors were problematic: they referenced the year 1900 as a base, so creating a `Date` with the value `115` meant the year 2015. Months were zero-based (e.g., 4 for May), but days were one-based, making `new Date(115, 4, 6)` a cryptic way to represent May 6, 2015. This constructor is deprecated now, and rightly so—it's very confusing and error-prone.

```
Date d = new Date(115,4,6);                                                    ①
String s = d.toString();                                                       ②
```

① Deprecated
② "Wed May 06 00:00:00 CEST 2015"

When instantiated with no arguments `Date` provides the current date. But even that relies on the time zone of the system it is running on.

> **Note**
>
> While older classes like `Date` are still found in legacy code, they are not recommended. Most of their methods are deprecated, and they're not well-suited for complex operations. The newer `java.time` classes are much more robust, consistent, immutable, and suitable for any kind of date/time processing.

13.2.2. Calendar

To improve this, Java introduced – in version 1.1 – an abstract class `Calendar` and its primary implementation, `GregorianCalendar`. This class provides standard fields for date components (e.g., `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR`, …) and lets you get and set these fields. You can also perform operations like adding days, months, or weeks—operations that are calendar-aware.

The main method for interacting with `Calendar` are:

- `get(field)`
- `set(field, value)`
- `add(field, delta)`

13.3. New Date and Time

Starting with Java 8, a new date/time API was introduced in the `java.time` package, which provides a comprehensive and consistent set of classes designed according to a few guiding principles:

- Simplicity
- Consistency
- Immutable classes

The classes and interfaces fall into two main categories:

- **Temporal points**: such as `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime`. These represent moments in time, either in absolute terms or relative to a time zone.
- **Temporal amounts**: such as `Duration` (time-based intervals) and `Period` (date-based intervals).

13.3.1. Time points

These types are immutable and do not expose public constructors. Instead, they use factory methods like `of()` or `parse()`. This applies uniformly—for instance, `LocalDate.of(2025, 5, 6)` clearly constructs May 6, 2025, without any weird base-year offsets.

Method	Purpose
`of()`	Create instance from a set of specific parameters, with validation
`from()`	Convert from another class with possible loss of information
`parse()`	Parse a string to build an instance
`now()`	Create an instance representing the current time / date. Can accept a ZoneId

Comparison

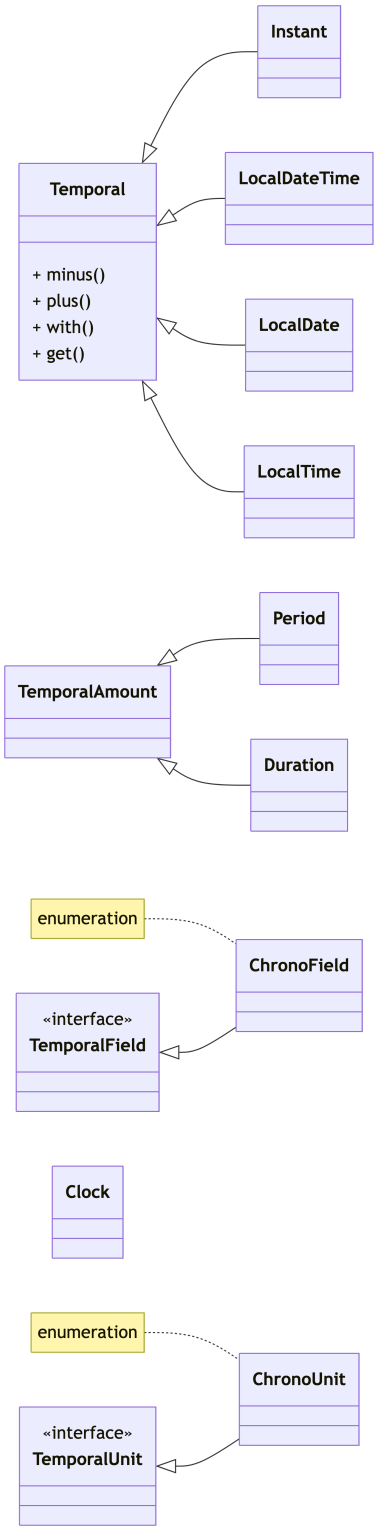


Figure 13.1.: Main classes of datetime API

Method	Purpose
`isBefore()`	Checks if this time/date is before the specified time/date
`isAfter()`	Checks if this time/date is after the specified time/date
`isEqual()`	Checks if this time/date is the same as the specified time/date
`compareTo()`	Compares to to other time/date

13.3.2. Changing

Method	Purpose
`minus()`	Returns a new date/time built by removing a specific amount of date/time
`plus()`	Returns a new date/time built by adding a specific amount of date/time
`with()`	Returns a new date/time modified as specified by a temporal adjuster

To manipulate dates or times, methods like `plus()` and `minus()` are available. These come in two flavors:

1. You pass a `long` and a `ChronoUnit` (e.g., `DAYS`, `MONTHS`, `YEARS`).
2. You use a `TemporalAmount` like a `Period` or `Duration`.

You can also use **TemporalAdjusters** to adjust a date to meaningful calendar positions thought method `with()`, for instance:

- `firstDayOfMonth()`
- `firstDayOfNextMonth()`
- `firstInMonth(DayOfWeek dayOfWeek)`
- `lastDayOfMonth()`
- `next(DayOfWeek dayOfWeek)`
- `previous(DayOfWeek dayOfWeek)`

In addition there are class specific method, e.g., `plusDays(long toAdd)`.

Day of Week and Month are represented by enums:

- `DayOfWeek`
- `Month`

Can be converted to string using: `getDisplayName(style,locale)`, where style is one of:

- `TextStyle.FULL`
- `TextStyle.NARROW`
- `TextStyle.SHORT`

Examples:

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1,ChronoUnit.DAYS);
LocalDate inTwoWeeks = today.plusDays(14);
LocalDate lastMonday = oggi.with(TemporalAdjusters.previous(DayOfWeek.MONDAY));
```

13.3.3. Locale

Class `Locale` represents a specific geographical, political, or cultural region Used to perform locale-sensitive operations:

- Date formats
- DoW, Month names
- Decimal separators
- Locale definition

Predefined constants, e.g., `Locale.US`, `Locale.ITALIAN`

Constructors

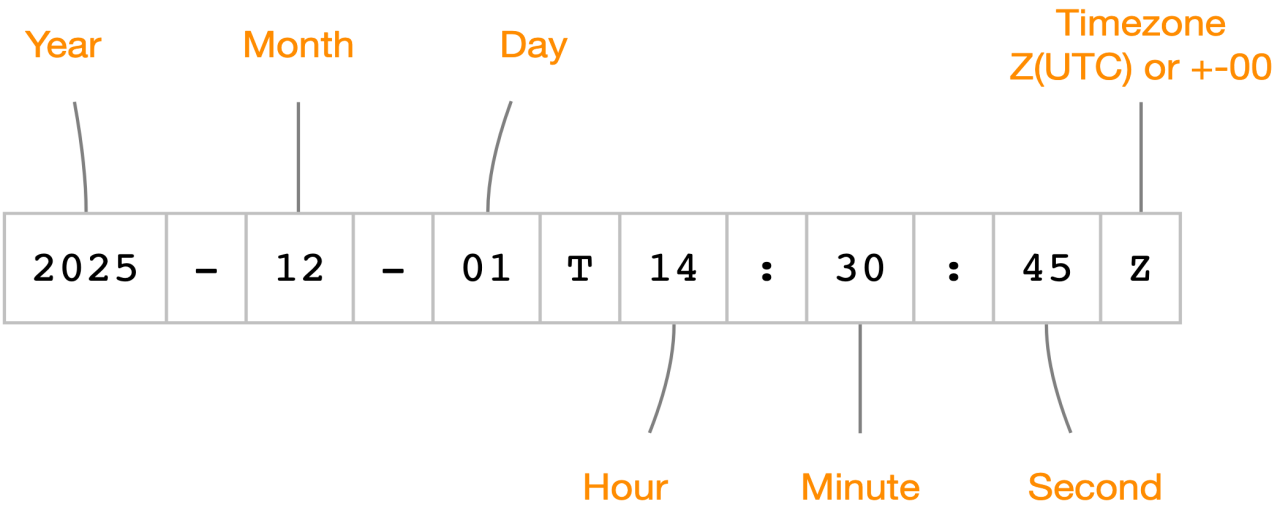
- Language: 2 or 3 chars code
- Country: 2 chars or 3 digits

In addition a variant allows optional additional specifications.

13.3.4. ISO-8601

A general recommendation is to use the ISO 8601 date format: `YYYY-MM-DD`, with four-digit years and two-digit months and days. This standard avoids ambiguity—unlike the European format (`DD/MM/YYYY`) or the U.S. format (`MM/DD/YYYY`), which can easily be misinterpreted.

The full ISO 8601 format also supports date and time, separated by a `'T'` (e.g., `2025-06-06T15:30:00+02:00`). The time zone is included as either `'Z'` for UTC or a `+/-` offset.



ISO 8601 Date and time format

13.3.5. Intervals

For time intervals, use `Duration` and `Period`. You can create these with factory methods or with `between(start, end)`, which calculates the elapsed time between two points.

Intervals can be created with the folowing factory methods:

Method	Purpose
`of()`	Creates interval from specified amount of TemporalUnits
`ofXxxx()`	Creates interval from specified amount of units (Xxxx is : Days, Hours, etc.)
`between()`	Creates interval between two temporal points

Example: measuring elasped time for

```
Instant start = Instant.now();
// here some long running operation
Instant end = Instant.now();
Duration elapsed = Duration.between(start, end);
IO.println(elapsed);
```

This prints a result like `PT0.003S` (3 milliseconds), using ISO 8601 duration format, where `PT` stands for Period of Time.

13.4. Testing

Testing code that is time dependent can be difficult. If in code that depend on time (e.g., booking an exam session) `now()` is hardcoded, it makes testing difficult.

For this, Java provides the `Clock` class, which you can pass to time-based methods. A `Clock` can be fixed to a specific instant or offset from the actual system time, making it ideal for testing. A clock object can be used as argument of `now()`.

Clocks can be created with the following factory methods:

- `fixed(instant, zone)` returns a clock that always returns the same instant, it is mainly used to testing purposes.
- `offset(base, offset)` returns a clock yields instants from the specified clock with the specified duration added, used to simulate past or future events w.r.t. a the given base clock.
- `systemDefaultZone()` obtains a clock aligned with the system clock in the system time zone, be careful since it implies a dependency to the default time-zone into your application.
- `systemUTC()` obtains a clock aligned with the system clock in the UTC time zone.

As a simple example of method that has dependency on the current time is the computation of a total amount due to return an initial capital plus a monthly rate, given the initial date.

Given an initial sum (*amount*) and a monthly interest rate ($0 < rate < 1$), the total due (*due*) can be computed using the compount interest rate after a given number of months (*months*):

$$due = amount \cdot (1 + rate)^{months}$$

An example of code used to compute the amount due today is:

```
static double totalDue(double amount, LocalDate begin, double monthlyRate){
  LocalDate today = LocalDate.now();

  Period interval = Period.between(begin, today);
  int months = interval.getMonths();
  double compoundRate = Math.pow(1.0+monthlyRate, months);
  return amount*compoundRate;
}
```

Of course this method will return a different value based on the day of execution of the test, thus making testing more difficult.

A testable version of the previous date-based computation should include a `Clock` in the computation that enable faking the date of execution:

```
static Clock clock = Clock.system();
static double totalDue(double amount, LocalDate begin, double monthlyRate){
  LocalDate today = LocalDate.now(clock);
  Period interval = Period.between(begin, today);
  int months = interval.getMonths();
```

```
  double compoundRate = Math.pow(1.0+monthlyRate, months);
  return amount*compoundRate;
}
```

With the testable version it is possible to write the following test

```
@Test
public static void testTotalDue(){
  LocalDate begin = LocalDate.of(2025,4,10);                    ①
  double r = 0.01;
  int amount = 1000;

  LocalDate in4 = begin.plusMonths(4));                         ②
  clock = Clock.fixed(in4.atStartOfDay(zone).toInstant(),       ③
                      ZoneId.systemDefault());

  double t = totalDue(amount, begin, r);                        ④

  assertEquals(amount*Math.pow(1+r, 4), t, 1);
}
```

① the values of the arguments to test the method
② compute the day four mounths after the begin date
③ create a fixed clock with the latter date
④ call the method regularly

Wrap-up

- Old `Date` class does not handle time zones correctly. If all you need is a simple date container, storing year/month/day in a custom class may be fine.
- If you plan to calculate time intervals or work with time zones, using the `java.time` API is strongly advised.
- New classes provide a consistent structure for both time and date measures:
  - They are immutable
  - Operations can be performed using existing methods
- Testing time and date based operations can be complex, the use of `Clock` is advised to make them testable

14. Input-Output (Draft)

All I/O operations rely on the abstraction of stream (flow of elements). While the term *stream* is in commont the I-O steram just the abstract idea is shared with Stream API.

14.1. I/O Stream

An I/O stream can be linked to:

- A file on the disk
- Standard input, output, error
- A network connection
- A data-flow from/to different hardware devices

I/O operations work in the same way with all stream from any source

All I/O stream classes and interfaces are defined in package: `java.io`. There are two main families of streams:

- Streams of chars with main classes `Reader` and `Writer` used to handle all text contents
- Streams of bytes with the main classes `InputStream` and `OutputStream`, used to handle binary data, e.g., sounds, images, videos.

All related exceptions are subclasses of `IOException`

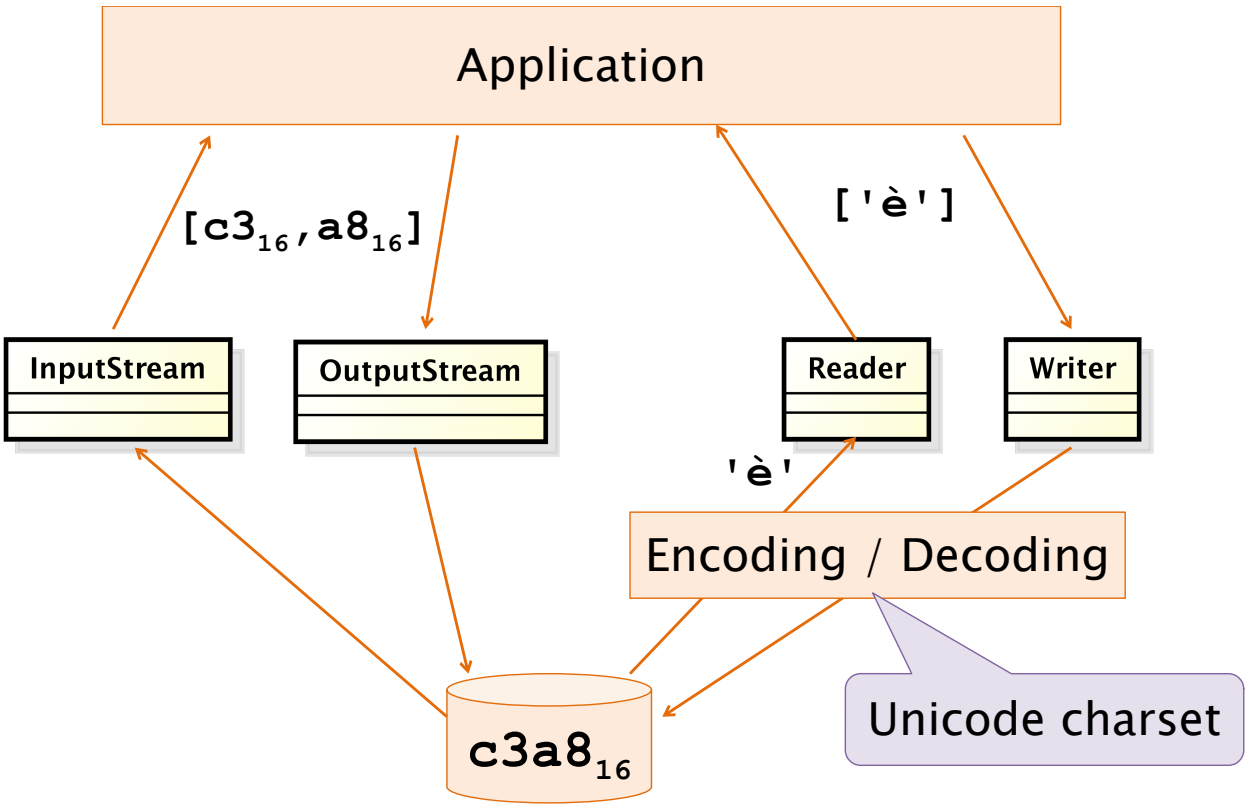


Figure 14.1.: Byte vs. Char Oriented Streams

14.1.1. Stream specializations

- Memory: R/W chars from/to array or String
  - CharArrayReader
  - CharArrayWriter
  - StringReader
  - StringWriter
  - ByteArrayInputStream
  - ByteArrayOutputStream
- Pipe Pipes are used for inter-thread communication they must be used in connected pairs
  - PipedReader
  - PipedWriter
  - PipedInputStream
  - PipedOutputStream
- File Used for reading/writing files
  - FileReader
  - FileWriter
- Buffered
- Printed

- Interpreted

14.2. Character Oriented Streams

14.2.1. Readers

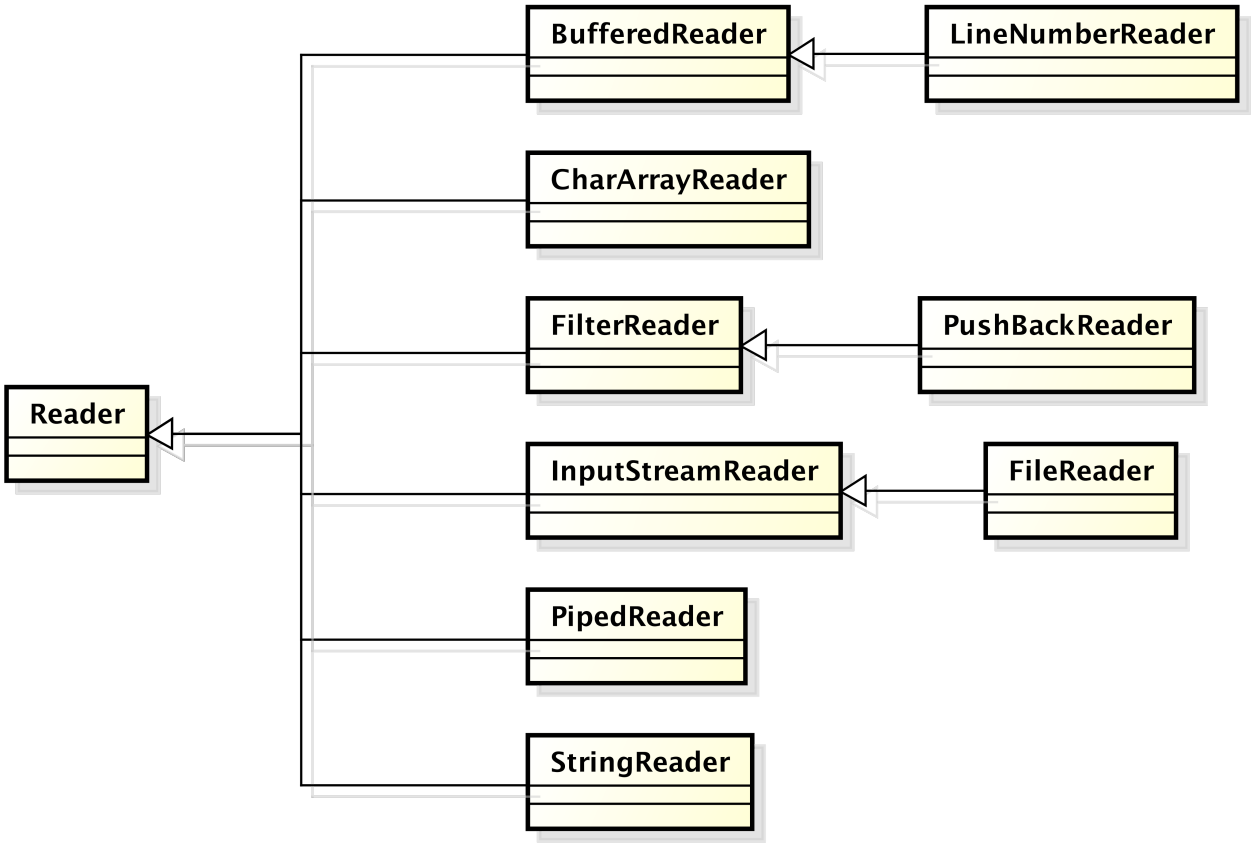


Figure 14.2.: Reader classes

Reader (abstract)

- void close() Close the stream.
- int read() Read a single character: returns -1 when end of stream
- int read(char[] cbuf) Read characters into an array.
- int read(char[] cbuf, int off, int len) Read characters into a portion of an array.
- boolean ready() Tell whether the stream is ready to be read.
- void reset() Reset the stream, restart from beginning
- long skip(long n) Skip n characters

Read a subgkle character

```
int ch = r.read();
char unicode = (char) ch;
System.out.print(unicode);
r.close();
```

Character	`ch`	`unicode`
`A`	$0 \ldots 0000000001000001_{bin} = 65_{dec}$	65
`\n`	$0 \ldots 0000000000001101_{bin} = 13_{dec}$	13
End of file	$1 \ldots 1111111111111111_{bin} = -1_{dec}$	-

Read a line

```
public static String readLine(Reader r) throws IOException{
  StringBuffer res= new StringBuffer();
  int ch = r.read();
  if(ch == -1) return null; // END OF FILE!
  while( ch != -1 ){
    char unicode = (char) ch;
    if(unicode == '\n') break;
    if(unicode != '\r') res.append(unicode);
    ch = r.read();
  }
  return res.toString();
}
```

14.2.2. Writers

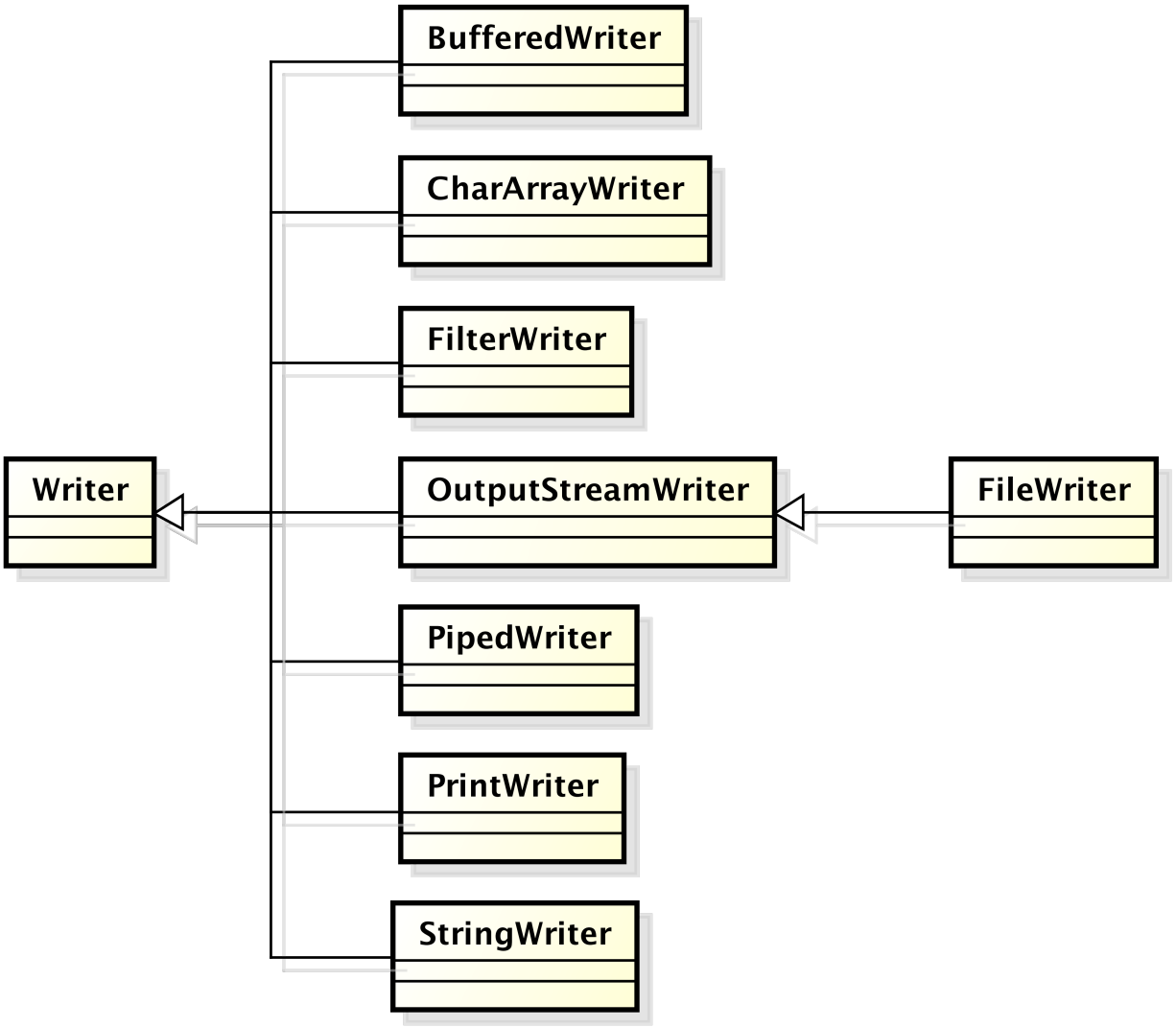


Figure 14.3.: Writer classes

Writer (abstract)

- void write(int c) Write a single character.
- void write(char[] cbuf) Write an array of characters.
- void write(char[] cbuf, int off, int len) Write a portion of an array of characters.
- void write(String str) Write a string.
- close() Close the stream, flushing it first.
- abstract void flush() Flush the stream.

Output streams typically write to a memory buffer Much faster than writing e.g. to file (leverage memory hierarchy) When buffer is full a large chunk is written, as a whole, to the destination Program termination wipes buffers Programmer must explicitly ensure buffers are flushed using method `close()` or `flush()`

14.3. Byte Oriented Streams

14.3.1. Input streams

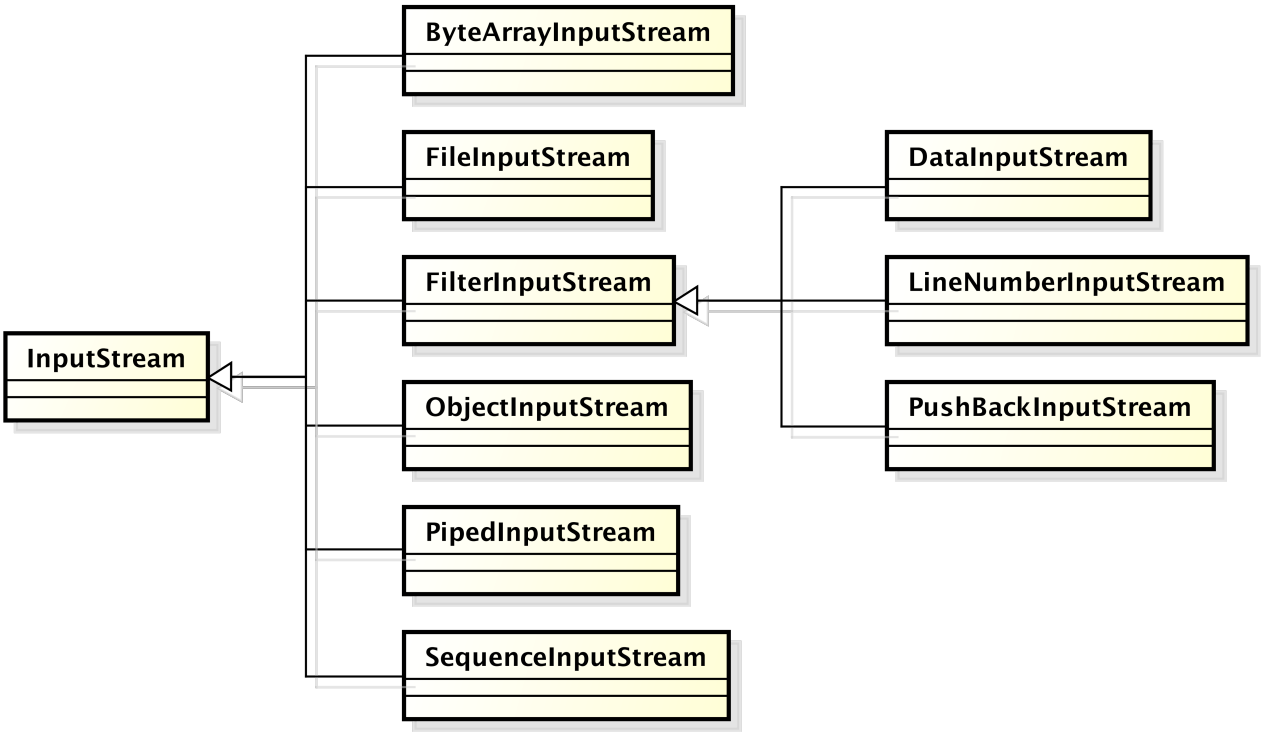


Figure 14.4.: InputStream classes

Class `InputStream`

- void close() Closes this input stream and releases any system resources associated with the stream.
- int read() Reads the next byte of data from the input stream.
- int read(byte[] b) Reads some bytes from the input stream and stores them into the buffer array b.
- int read(byte[] b, int off, int len) Reads up to len bytes of data from the input stream into an array of bytes.
- int available() Number of bytes that can be read (or skipped over) from this input stream without blocking.
- void reset() Repositions this stream to the position at the time the mark method was last called.
- long skip(long n) Skips over and discards n bytes of data from this input stream.

14.3.2. Output streams

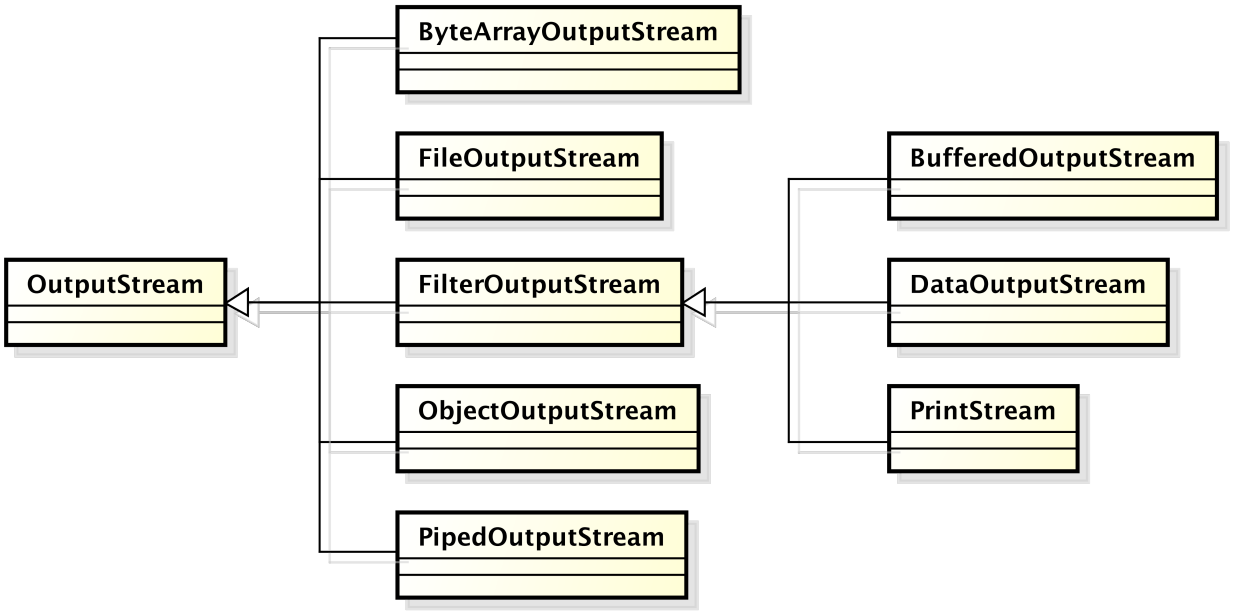


Figure 14.5.: OutputStream classes

Class `OutputStream`

- void write(byte[] b) Writes b.length bytes from the specified byte array to this output stream.
- void write(byte[] b, int off, int len) Writes len bytes from the specified byte array starting at offset off to this output stream.
- void write(int b) Writes the specified byte to this output stream.
- void close() Closes this output stream and releases any system resources associated with this stream.
- void flush() Flushes this output stream and forces any buffered output bytes to be written out.

14.3.3. File streams

Copy text file

```
Reader src = new FileReader(args[0]);
Writer dest = new FileWriter(args[1]);
int in;
while( (in=src.read()) != -1){
    dest.write(in);
}
src.close();
dest.close();
```

Copy text file with buffer

```
Reader src = new FileReader(args[0]);
Writer dest = new FileWriter(args[1]);
char[] buffer = new char[4096];
int n;
while((n = src.read(buffer))!=-1){
  dest.write(buffer,0,n);
}
src.close();
dest.close();
```

14.3.4. Text file with encoding

- `InputStreamReader` byte to char
- `OutputStreamWriter` char to byte

The constructors allow specifying a charset to decode/encode the byte to/from characters

The text encoding of a stream can be defined using InputStreamReader for input

```
Reader r = new InputStreamReader(new FileInputStream("file.txt"), "ISO-8859-1");
```

Since Java 11 a new constructur for class `FileReader`

```
Reader r = new FileReader("file.txt", Charset.forName("ISO-8859-1"))
```

OutputStreamWriter for output

```
Writer w = new OutputStreamWriter(new FileOutputStream("out.txt", "ISO-8859-1"));
```

Since Java 11 a new constructur for class `FileWriter`

```
Writer w = new FileWriter("file.txt", Charset.forName("ISO-8859-1"))
```

14.3.5. Buffered

`BufferedInputStream` BufferedInputStream(InputStream i) BufferedInputStream(InputStream i, int s)

`BufferedOutputStream`

`BufferedReader` readLine()

`BufferedWriter`

14.3.6. Printed streams

Class `PrintStream`

- `PrintStream(OutputStream o)` Provides general printing methods for all primitive types, String, and Object
- print() and println()

Designed to work with basic byte-oriented console

Does not throw `IOException`, but it sets a bit, to be checked with method `checkError()`

Default input and output streams are defined in class `System` using printed streams

```
class System {
  //…
  static InputStream in;
  static PrintStream out;
  static PrintStream err;
}
```

Default streams can be replaced `setIn()`, `setOut()`, `setErr()`

Example:

```
String input = "This is\nthe input\n";
InputStream altInput = new ByteArrayInputStream(input.getBytes());
InputStream oldIn = System.in;
System.setIn(altInput);
readLines();
System.setIn(oldIn);
```

14.3.7. Interpreted streams

Translate primitive types into / from standard format Typically on a file

- `DataInputStream(InputStream i)`
  - readByte(), readChar(), readDouble(), readFloat(), readInt(), readLong(), readShort(), ..
- `DataOutputStream(OutputStream o)` like write()

14.3.8. Streams and URLs

Streams can be linked to URL

```
URL page = new URL(url);
InputStream in = page.openStream();
```

Be careful about the type of file you are downloading.

Class `URL` Represents a URL Constructor may throw a `MalformedURLException`

Provide getters for URL portions: Protocol, Host, Port Path, Query, Anchor (Ref)

Method `openStream()` opens a connection and returns the relative `InputStream`

Download a file

```
URL home = new URL("http://…");
URLConnection con = home.openConnection();
String ctype = con.getContentType();
if(ctype.equals("text/html")){
  Reader r = new InputStreamReader(
                                        con.getInputStream());
  Writer w = new OutputStreamWriter(System.out);
  char[] buffer = new char[4096];
  while(true){
    int n = r.read(buffer);
    if(n==-1) break;
    w.write(buffer,0,n);
  }
  r.close(); w.close();
}
```

14.3.9. Stream as Resources

Streams consume OS resources and and such resources are limited. In general a process can open only a limited number of files at once. For this reason, IO streams should be closed as soon as possible to release resources.

This is the reason why all classes implement the `AutoCloaseable` interfaces so that they can be used with the *try-with-resorce* construct.

14.4. Serialization

Serialization is the procedure through which an object in memory can be transformed into a sequence of bytes that store its current state. Such a sequence of bytes can be written to any storage and retrieved later. The inverse process of **Deserialisation** is able to convert the sequence of bytes into an object in memory that has the same state as the original one.

Read / write of an object implies to read/write all the attributes of the object, and with their types. In particular a proper structure should be used to correctly separating different elements. When reading, a new object must be created and all its attributes values restored.

These operations (serialization) are automated by - `ObjectOutputStream` serialization - though method `void writeObject(Object)` - `ObjectInputStream` deserialization - through methdo `Object readObject()`

It is possible to serialize only objects implementing the interface `Serializable`. This interface is empty and works as a and flagging interface. This constraint avoid serialization of objects, without permission of the class developer. The implementation of the interfaces flags the classes that can meaningfully serialized.

A critical problem in deserialization is type recovery: when reading, an object is created, but which is its type? In practice, not always a precise downcast is required: only if specific methods need to be invoked. A downcast to a common ancestor can be used to avoid identifying the exact class.

Serialization is applied recursively to object pointed to by reference attributes. Of course the referenced objects must implement the `Serializable` interface. Specific fields can be excluded from serialization by marking them as `transient`.

An `ObjectOutputStream` saves all objects referred by its attributes objects serialized are numbered in the stream. References are saved as ordering numbers in the stream. If two saved objects point to a common one, this is saved just once Before saving an object, `ObjectOutputStream` checks if it has not been already saved otherwise it saves just the reference

Serialization example

```
public class Student implements Serializable {
    \\ ...
}
```

Serializing (writing)

```
List<Student> students=new LinkedList<>();
students.add( /*...*/ );
// ...
ObjectOutputStream serializer = new ObjectOutputStream(new FileOutputStream("std.dat"));
serializer.writeObject(students);
serializer.close();
```

Deserializing (reading)

```
ObjectInputStream deserializer = new ObjectInputStream(new FileInputStream("std.dat"));
Object retrieved = deserializer.readObject();
deserializer.close();
List<Student> l = (List<Student>)retrieved;
```

14.5. File System

Class `File` represent the concept of an abstract pathname, with:

- directory, file, file separator
- absolute, relative
- convert abstract pathname <–> string

The main methods:

- create() delete() exists() , mkdir()
- getName() getAbsolutePath(), getPath(), getParent(), isFile(), isDirectory()
- isHidden(), length()
- listFiles(), renameTo()

Example: list the files contained in the current working folder

```
File cwd = new File(".");
for(File f : cwd.listFiles()){
  System.out.println(f.getName()+ " " + f.length());
}
```

14.6. New IO

New IO (`java.nio`)

- Paths and Files
- Abstract path manipulation
- Static methods
- Buffer and Channels
- Buffer oriented IO
- Leverages efficient memory transfers (DMA)

14.6.1. Class `Path`

Represents path in the file system

Build from a string or URI `of(String first, String… more)`

Return the corresponding path

Components extraction:

- getFileName()
- getName(int index)
- getParent()
- getRoot()

Relative paths

- isAbsolute()
- relativize(Path other)
- resolve(Path other)

```
Path home = Path.of("/home/mtk");
Path maven = home.resolve(".m2");                                    ①
Path relMaven = home.relativize(maven);                              ②
```

① `/home/mtk/.m2`
② `.m2`

14.6.2. Class `Files`

Provides methods to operate on Paths

Copy content:

- copy(Path , Path, CopyOptions…)
- move(Path , Path, CopyOptions…)

Create:

- createFile(Path)
- createDirectory(Path,FileAttribute…)

Test properties:

- isWritable(Path)
- isDirectory(Path)

Navigate return a `Stream<Path>`

- list()
- find(Path start, int maxDepth, BiPredicate<> matcher, FileVisitOption… options)

Read:

- Stream lines(Path)
- List readAllLines(Path)
- String readString(Path)

Write:

- write(Path, String)

- write(Path, Iterable)

Example Compute max line length

```
Path p = Paths.get("file.txt")
int maxLen = 0;
if(Files.exists(p)){
    maxLen = Files.lines(p).
                mapToInt(String::length).
                max().getAsInt();
}
```

14.7. Wrap-up

- Java IO is based on the stream abstraction
- Two main stream families:
  - Char oriented: Reader/Writer
  - Byte oriented: Input/OutputStream
- There are streams specialized for Memory, File, Pipe, Buffered, Print
- Streams resources need to be closed as soon as possible
- Try-with-resource construct guarantee resource closure even in case of exception
- Serialization means saving/restoring objects using Object streams
- Serializable interface enables it

15. Object-Relational Mapping (ORM) (Draft)

The persistence of complex data structures in a program can be achieved using a data-base. The Object-Oriented model and the Relational models, while similar in some aspects have a few incompatible aspects – usually called mistmatches) – that make the integration difficult. While low-level interaction through queries is possible – using the Java DataBase Connectivity (JDBC) API – modern applications adopt an adaptation layer called **Object-Relational Mapping (ORM)** that makes code simpler and allow flexibility and optimization.

15.1. Persistence and Direct data access

15.1.1. Persistence

Persistence is the property of data that can outlive the current process lifespan, i.e. when a program is started it will have data from previous work sessions available.

For Java developers, this means we would like the state of certain objects to live beyond the scope of the JVM, so that the same state is available in a different JVM, or even in a different program written in a different programming language.

The key requirement for persistence is that data must survive program execution. Simple program use *in-memory data* which is stored in RAM and lost when the program stops. Persistent data need to be stored on disk, databases, or external storage device. The key differences are: volatility, lifespan, and accessibility. In-memory data is volatile and disappears when power is lost or the application terminates, while persistent data remains intact across sessions. The lifespan of in-memory data is limited to the duration of program execution, whereas persistent data can exist indefinitely. Finally, in-memory data is accessible only within the running process, while persistent data can be shared across different applications, users, and even platforms, making it essential for long-term data storage and multi-user systems.

15.1.2. Object/Relational Mismatch

The incompatibilities between the relational data model compared object-oriented paradigm are usually known ad object/Relational Mismatch or *impedance mismatch*:

- the relational model represents data in a tabular format with references based on values,
- object-oriented languages represent data as an interconnected graph of objects with inheritance and polymorphism.

The most important mismatches consist in:

- **Granularity** Object models often have more classes than tables in the database.

- **Subtypes (inheritance)** The relational model doesn't have inheritance as a first-class construct, There are standard ways to represent subtyping.
- **Identity** The relational model defines exactly one notion of "sameness": the primary key. Java defines both object identity a==b and value equality a.equals(b). It is common to have more than one object in a given Java program representing the same row of a database table.
- **Associations** Associations are represented as unidirectional references in Java Relational databases treat all associations as bidirectional and represent them via foreign keys Bidirectional relationship in Java require code on both sides of the association Just by looking at the Java code we cannot be sure of the multiplicity of an association
- **Data navigation** In Java, we navigate associations from one object to the next, walking the object network. That is inefficient when retrieving relational data The goals is to minimize the number of SQL queries

15.1.3. Object-Relational Mapping (ORM)

Object-Relational Mapping (ORM) is a technique for managing database interactions using object-oriented programming, converting data between a relational database and the memory.

An ORM solution:

- provides an abstraction layer that decouples business logic from database access,
- translates objects in the application into database records and vice-versa,
- automates data conversion between object models and relational schemas,
- reduces (in most cases eliminates) the need for direct SQL queries for basic operations,
- automates query generation and result mapping to objects,
- reduces manual connection and transaction management.

The alternative to ORM is Direct Data Access through the JDBC API:

- **Direct Data Access**
  - Uses native SQL to interact with the database
  - Directly embeds SQL in the application code
  - Manually handles every relationship and transaction
- **ORM**
  - Uses object-based operations to interact with the database
  - SQL queries can be automatically generated based on object models
  - Simplifies complex relationships and data handling

ORM Pros:

- Reduces the need to insert SQL within our programming language
- Improves code maintainability and readability
- Reduces development time by automating database interactions
- Supports multiple databases with minimal changes

ORM Cons:

- Less control over query optimization and performance (but it is possible to use direct SQL queries)

- Potential overhead compared to manually optimized SQL
- Learning curve for ORM frameworks and configurations

15.2. JPA Mapping

Jakarta Persistence API (JPA), formerly Java Persistence API, is a Java specification for managing relational data using an object-oriented approach. It defines a standard way to map Java objects to database tables, manage entity lifecycle, and execute queries.

JPA itself does not provide an implementation; it relies on providers such as Hibernate to handle persistence operations.

JPA provides two approaches for defining entity mappings: annotation based (modern approach) and XML-based (mostly for legacy systems), in this chapter we will focus on annotations approach.

JPA annotations are applied at different levels:

- Entity-Level: defines an entity and its table mapping
- Field-Level: defines the mapping of individual fields to database columns
- Relation-Level: defines associations between entities

15.2.1. Entity

An entity is a purely structural component that represents a persistent object that is mapped to a database table. It is a fundamental building block of the application's domain model and serves as an abstraction over relational database records.

Entities encapsulate only data structure and constraints, without containing business logic. Each entity instance corresponds to a row in a relational table. An entity defines fields (mapped to table columns) and relationships with other entities. Must have at least one primary key to ensure uniqueness.

By defining entities, we model the granularity of the object domain, ensuring a structured representation of data in the application.

The following annotations at the class level can be used to define entity mapping:

- `@Entity` Marks a class as a JPA entity.

  The class must have a no-argument constructor with at least package visibility.

  If not specified, the table name defaults to the class name.

- `@Table(name, indexes...)` Specifies the database table on which the entity will be mapped
  - `name` (`String`, optional): the name of the db table
  - `indexes` (`Index[]`, optional): array of the indexes of the table

- `@Index(name, columnList, unique)` Indexes are defined within the `@Table` annotation
  - `name`: (`String`) the name of the index
  - `columnList`: (`String`) comma-separated list of column names that will be included in the index.

  - `unique`: (`boolean`, default: false) whether the index enforces uniqueness across all indexed columns. If multiple columns are indexed together, their combination must be unique

- `@Inheritance(strategy)` specifies the inheritance strategy for an entity hierarchy

  - `strategy` (Enum: `InheritanceType`):

  Relational databases do not support subtypes (inheritance) natively, so ORM provides different strategies to map class hierarchies to tables. There are three main mapping strategies:

  - Single Table Inheritance (`SINGLE_TABLE`) – One table stores all subclasses, with a type discriminator column, this is the most efficient strategy but wastes space;
  - Joined Table Inheritance (`JOINED`) – Each class has its own table, linked through foreign keys, normalizes data but adds complexity;
  - Table Per Class (`TABLE_PER_CLASS`) – Each class has a separate table with all its attributes; avoids duplication but makes querying harder.

- `@DiscriminatorColumn(name, discriminatorType, length)` Used with `SINGLE_TABLE` inheritance to distinguish entity types

  - `name` (`String`, default: `"DTYPE"`): name of the discriminator column
  - `discriminatorType` (Enum: `DiscriminatorType`): type of the column `STRING` | `CHAR` | `INTEGER`,
  - `length` (`Integer`, default: 31): max length for STRING type

- `@DiscriminatorValue(value)` Used with `SINGLE_TABLE`, defines the value stored in the discriminator column for a specific entity

  - `value` (`String`): discriminator value for this entity

- `@MappedSuperclass`. Defines a non-entity superclass, the class itself does not correspond to a database table.

  It can be used to share mapped fields among subclasses. Its annotated fields (`@Id`, `@Column`, etc.) are inherited and mapped into the tables of concrete entity subclasses. Ideal for shared attributes such as id, timestamps, audit fields, or other common metadata. With this annotation, inheritance exists only on the Java side, there is no inheritance structure in the database —- i.e. no join, no discriminator column –.

15.2.2. Fields

Each entity field corresponds to a database column and can have the constraints and properties typical of fields in a relational database schema.

Data type conversion between object model and relational model:

- `String` in code → `VARCHAR` in database
- `boolean` in code → `TINYINT(1)` in some databases

Some languages support richer types than relational databases, ORM provides mapping rules for correct conversion. Custom type mapping possible if default conversions are not sufficient.

The fields can be annotated with the following annotations:

- `@Id` Declares the field as the *primary key (PK)* that establish the identity of an entity instance; it defines entity uniqueness. Can be auto-generated (e.g., auto-increment, UUID) or manually assigned.

- `@GeneratedValue(strategy)` Define an a PK (marked with `@Id`) as autogenerated. It can specify which different generation `stategy` (Enum: `GenerationType`) is adopted:

  - `AUTO`: default, ORM decides the best strategy
  - `IDENTITY`: uses database auto-increment
  - `SEQUENCE`: uses a database sequence
  - `TABLE`: uses a table-based sequence

- `@Column(name, nullable, length, unique, insertable, updatable)` Details the mapping of a field to a database column, overriding the default one:

  - `name` (`String`, optional): the name of the column in the mapping table
  - `nullable` (`boolean`, default: false): if true, allows null values
  - `length` (`int`, default: 255): defines the maximum length for a String field
  - `unique` (`boolean`, default: false): if true, enforces a unique constraint
  - `insertable` (`boolean`, default: true): if false, prevents insertion
  - `updatable` (`boolean`, default: true): if false, prevents updates

- `@Transient` Excludes a field from persistence, thus the field will not be mapped to a column. By default all attributes in an entity class are mapped to fields in the corresponding table.

- `@Enumerated(value)` Maps an enumerative type to a database column. The parameter `value` defines how defines how it is stored in the db:

  - `ORDINAL`: stores enum positions (0,1,2…)
  - `STRING`: stores enum names as strings

- `@Lob` Marks a field as a large object (BLOB/CLOB).

- `@Temporal(value)` Specifies the temporal type of a `java.util.Date` or `java.util.Calendar`. Parameter `value`: (Enum: TemporalType) defines how the date/time is stored

  - `DATE`: Stores only the date (without time).
  - `TIME`: Stores only the time (without date).
  - `TIMESTAMP`: Stores both date and time

15.2.3. Relations

Relationships define the associations of the object-oriented model, how entities are linked in the relational model.

Four main types of relationships:

- One-to-One (1:1) (`@OneToOne`): each instance of the entity is associated with exactly one instance of the related entity;
- Many-to-One (N:1) (`@ManyToOne`): multiple instances of the entity are associated with a single instance of the related entity;
- One-to-Many (1:N) (`@OneToMany`): one instance of the entity is related to multiple instances of the related entity;
- Many-to-Many (N:N) (`@ManyToMany`): entities are related in a many-to-many fashion, requiring a join table.

JPA manages relationships using foreign keys and ensures data consistency using entity associations. Foreign keys are automatically managed by JPA based on relationship annotations, customization is allowed by using `@JoinColumn` (One-to-One, Many-to-One) or `@JoinTable` (Many-to-Many).

When defining relationships it is often useful to distinguish between **Owning** vs. **Inverse** side:

- the **owning** side is the entity that stores the foreign key,
- the **inverse** side is the entity that is referenced.

Another importan distinction is between Bidirectional vs. Unidirectional relationships:

- **Unidirectional**: only the owning side is aware of the relationship
- **Bidirectional**: both sides are aware of the relationship; the inverse side uses `mappedBy` to refer to the field on the owning side.

The fields that implement an association and will be mapped to a relationship in the relational model can be annotated with the following annotations:

- `@OneToOne(cascade, fetch, mappedBy, orphanRemoval, optional)`
  - `cascade` (Enum: CascadeType[], optional): defines the cascading strategy
  - `fetch` (Enum: FetchType, optional, default: EAGER): defines the fetch strategy
  - `mappedBy` (String): specifies the field in the owning entity that maps this relationship. Used only on inverse side
  - `orphanRemoval` (boolean, default: false): if true, removes related entities when they are no longer referenced
  - `optional` (boolean, default: true): if false, enforces a non-null foreign key constraint
- `@ManyToOne(cascade, fetch, optional)`
  - `cascade` (CascadeType[], optional): defines the cascading strategy
  - `fetch` (FetchType, optional, default: EAGER): defines the fetch strategy
  - `optional` (boolean, default: true): if false, enforces a non-null foreign key constraint
- `@OneToMany(mappedBy, cascade, fetch, orphanRemoval)`

  - `mappedBy` (String, required): specifies the field in the owning entity that manages the foreign key
  - `cascade` (Enum: CascadeType[], optional): defines the cascading strategy
  - `fetch` (Enum: FetchType, optional, default: LAZY): defines the fetch strategy
  - `orphanRemoval` (boolean, default: false): if true, removes orphaned entities automatically

  `@JoinColumn` cannot be used with `@OneToMany` because the foreign key is in the "many" side (`@ManyToOne`) It uses `mappedBy` to indicate that the relationship is managed by the "many" side

- `@ManyToMany(cascade, fetch)`

  - `cascade` (CascadeType[], optional): defines the cascading strategy
  - `fetch` (FetchType, optional, default: LAZY): defines the fetch strategy

  `@JoinTable` can be used to specify the table and foreign key mappings

- `@JoinColumn(name, referencedColumnName, nullable, unique, insertable, updatable)` used on the owning side with One-to-One, Many-to-One, to define the foreign key, by default, JPA automatically names the foreign key column as `{fieldName}_id`

  - `name` (String, optional): defines the foreign key column name
  - `referencedColumnName` (String, optional): the column name in the referenced entity's table
  - `nullable` (boolean, default: true): if false, prevents NULL values
  - `unique` (boolean, default: false): if true, enforces uniqueness
  - `insertable` (boolean, default: true): if false, prevents insert operations
  - `updatable` (boolean, default: true): if false, prevents updates

  It can be used on the owning side only, the inverse side of the relationship uses `mappedBy` to reference the owning entity and avoid an extra foreign key

- `@JoinTable(name, joinColumns, inverseJoinColumns)` Used with `@ManyToMany` relationships to define a join table, by default JPA automatically creates a join table

  - `name` (String, required): name of the join table
  - `joinColumns` (Array of `@JoinColumn`): defines foreign key columns for the owning entity
  - `inverseJoinColumns` (Array of `@JoinColumn`): defines foreign key columns for the inverse entity

15.2.3.1. Fetch strategy

The **Fetch** (or loading) strategy defines how the related entities are loaded and thus determines the data navigation in the model.

Navigating relationships in ORM requires fetching data efficiently. The two primary data loading strategies are:

- Lazy Loading (`LAZY`): related data is only loaded when explicitly accessed it improves initial performance but may cause multiple queries.
- Eager Loading (`EAGER`): related data is loaded immediately with the entity. It reduces query count but can load unnecessary data

Choosing the right strategy depends on performance trade-offs and data access patterns

15.2.3.2. Cascading strategy

Cascading allows propagation of operations (e.g., persist, update, delete) from a parent entity to its related entities.

ORMs let you configure cascading per operation type:

- Persist: saves related entities when the parent is saved.
- Merge: updates related entities when the parent is updated.
- Remove: deletes related entities when the parent is deleted.
- Detach: detaches related entities when the parent is removed from the persistence context.
- Refresh: reloads related entities when the parent is refreshed.

Cascade strategies:

- Full cascade: applies all operations to related entities.
- Selective cascade: allows only specific operations to be propagated.
- No cascade: relationships must be managed manually.

Cascading simplifies persistence management but must be configured carefully to avoid unintended side effects.

All relationship annotations share the:

- `cascade` (Enum: `CascadeType[]`): defines how persistence operations affect related entities. It accepts an array of CascadeType values, meaning you can specify different cascade behaviors for different operations:
  - `ALL`: applies all operations to related entities
  - `PERSIST`: saves child entities when the parent is saved
  - `MERGE`: updates child entities when the parent is updated
  - `REMOVE`: deletes child entities when the parent is deleted
  - `REFRESH`: reloads child entities when the parent is refreshed
  - `DETACH`: detaches child entities when the parent is detached

15.2.4. Example

```
@Entity
public class User {
    @Id
    @Size(min = 16, max = 16)
    private String cf;
    private String name;
    private Integer age;

    @OneToMany(mappedBy = "customer")
    private List<Order> orders = new ArrayList<>();

    User(){}
```

```
    // ...
}
```

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private User customer;

    private String status;

    Order(){}

    //...
}
```

15.3. JPA Persistence

15.3.1. Persistence Unit

JPA provides an abstraction layer over JDBC (Java DataBase Connectivity, the Java native DB connector), handling database connections and transactions. Entities are automatically managed in a persistence context, allowing seamless database interaction. JPA does not directly interact with the database; instead, it relies on a persistence provider (such as Hibernate) to execute queries and manage transactions

A Persistence Unit represents a logical configuration that groups:

- the database connection settings
- the list of managed entities
- transaction handling strategy
- JPA provider-specific settings

Each JPA application must declare at least one Persistence Unit. The Persistence Unit is the entry point for all JPA operations. The Persistence Unit is created when the application starts and remains available for database interactions.

`persistence.xml` file is the configuration file required to define how JPA interacts with the database. This file can include provider-specific properties that are not defined by JPA but are used to configure additional features of the chosen JPA provider.

Each Persistence Unit is defined by:

- a unique name
- the database connection settings (JDBC URL, driver, credentials)

- the transaction handling strategy (transaction-type):
  - `RESOURCE_LOCAL`: transactions are managed manually using `EntityTransaction`,
  - `JTA`: transactions are managed externally by a container.
- List of managed entities (optional if entities are defined with annotations and are in the classpath)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/
             version="3.1">
    <persistence-unit name="sample"> # <1>
        <properties>
          <property name="jakarta.persistence.jdbc.driver" # <2>
                    value="org.h2.Driver"/> # <2>
          <property name="jakarta.persistence.jdbc.url" # <3>
                    value="jdbc:h2:mem:sampledb;DB_CLOSE_DELAY=-1"/> # <3>
          <property name="jakarta.persistence.jdbc.user" value="sa"/> # <4>
          <property name="jakarta.persistence.jdbc.password" value=""/> # <4>
          <property name="jakarta.persistence.schema-generation.database.action" # <5>
                    value="drop-and-create"/> # <5>
        </properties>
    </persistence-unit>
</persistence>
```

① the unique name of the unit
② JDBC driver used for connection
③ JDBC connection string, with database name
④ database account details
⑤ Schema generation policy

ORM frameworks provide automatic schema generation policies based on the defined entities. Schema generation policy define how schema creation should be managed by the JPA provider. The possible policies are:

- `create`: Generates the schema if it does not exist
- `drop-and-create`: Deletes and recreates the schema on each run (useful for testing)
- `update`: Modifies the schema to match entity changes (may cause issues with existing data)
- `validate`: Checks if the schema matches the entity definitions but does not modify it
- `none`: The ORM does not perform any schema validation or modification

Choosing the right policy:

- Development: Create or Drop-and-Create for rapid iteration,
- Production: None or Validate to prevent unintended schema changes.

The `EntityManagerFactory` loads configuration from `persistence.xml` and initializes the JPA provider (e.g. Hibernate). It is a heavyweight object, meant to be created once per application and reused. It must always be closed when the application shuts down. It is responsible for managing persistence units and creating `EntityManager` instances

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit"); ①
EntityManager em = emf.createEntityManager();                                            ②
```

① loads the configuration of the persistence unit `myPersistenceUnit` defined in `persistence.xml`
② creates the `EntityManager` object that defines a session.

15.3.2. Session and `EntityManager`

A **Session** (or *persistence context*) is a temporary environment where an ORM tracks and manages entities during their lifecycle. It acts as an intermediary between the application and the database, ensuring that objects remain synchronized with their corresponding records in the database. Its scope and lifespan must be carefully managed to avoid performance issues.

Session key features:

- Manages the identity and lifecycle of entities within an application session
- Keeps track of changes made to managed entities
- Automatically synchronizes data between in-memory objects and the database
- Reduces the need for explicit SQL statements, leveraging automatic persistence
- Provides a caching mechanism, reducing unnecessary database queries
- Ensures consistency within a transaction, committing or rolling back changes

Session operations:

- Opening a session – Creates a new persistence context
- Persisting an entity – Registers a new entity for saving
- Fetching an entity – Retrieves an entity from the database into the session
- Updating an entity – Modifies an entity already tracked by the session
- Detaching an entity – Stops tracking an entity without deleting it
- Flushing changes – Writes all pending updates to the database
- Closing a session – Ends the persistence context and detaches all entities

When an entity is retrieved using an ORM query, it is automatically added to the session, its state is tracked, and any modifications will be synchronized with the database when the session is flushed or committed.

Entities transition through different states during their lifecycle in an ORM Main entity states:

- *Transient* – The entity is created in memory but not yet persisted
- *Persistent* – The entity is associated with a database session and will be saved
- *Detached* – The entity is no longer managed by the ORM but still exists in memory
- *Removed* – The entity is marked for deletion and will be removed from the database

These operations modify the state of an entity, when performed

- Creating a new instance (e.g. `new EntityClass()`) starts in *Transient* state
- Persisting the entity (e.g., `save()` or `persist()`) transitions it to *Persistent*
- Closing or clearing the session detaches the entity (*Detached*)
- Deleting the entity moves it to the *Removed* state

If the same entity is requested again during the same session, the ORM will return the existing in-memory instance rather than executing another database query.

When executing a native SQL query that retrieves data not directly mapped to a known entity, the results are not added to the session. In such cases, the ORM treats the output as raw data, and no automatic tracking or synchronization occurs.

Detaching an entity means removing it from the session while keeping it in memory. Once detached, the ORM no longer tracks changes made to the entity, and these modifications will not be reflected in the database unless the entity is reattached to a session.

Detaching is not removing. Removing an entity means marking it for deletion from the database. It is still tracked by the session until the session is flushed or committed, at which point the corresponding record is physically deleted from the database.

Detaching only affects the ORM's ability to track changes, while removing permanently deletes the entity's data from the database.

Detaching can be useful when dealing with large datasets to reduce memory consumption, whereas removing is necessary when an entity - and all its information - is no longer needed in the system.

`EntityManager` is the implementation of the concept of session and is the basis of all querying operations in JPA.

It is the main interface for interacting with the database in JPA. It manages the lifecycle of entities and executes queries using both JPQL and native SQL It is a lightweight, short-lived object, typically created per transaction or request When obtained directly from the `EntityManagerFactory`, it must be explicitly closed with `close()` to release resources.

It provides the core API for:

- Entity lifecycle management:
  - `persist(Object entity)`: registers a new entity to be inserted into the database when a transaction is committed
  - `merge(Object entity)`: updates an existing entity or reattaches a detached entity to the persistence context
  - `detach(Object entity)`: removes the given entity from the persistence context
  - `remove(Object entity)`: marks an entity for deletion from the database when the transaction is committed

  In addition there are two operation that allow synchronizing the in-memory content with the database:

  - `flush()`: write to the database all the pending changes without waiting for the transaction commit,
  - `refresh()`: updated the in-memory objects with the latest changes present in the database, possibly overwriting attributes.

  Note that any operation that modify data in the database requires an active transaction.
- Querying: supports different query methods allowing structured, reusable, dynamic, or raw SQL queries to retrieve data

  - `find(Class<T>, Object primaryKey)`: retrieves a single entity by its primary key

  For more complex queries, `EntityManager` provides methods that return a Query object, which allows customization and execution of different types of queries.

  Operations that read data do not require an active transaction

- Transaction management: explicitly controls transactions when a RESOURCE_LOCAL transaction type is used.

  - `getTransaction()`: creates a transaction object

Session can be managed as:

- short-lived: A new session is created for each operation (or web request) and closed immediately after. This ensures that entities are not kept in memory longer than necessary. Prevents excessive memory consumption and avoids issues such as long-running transactions. Recommended for most applications, especially web-based systems where each request is stateless.
- long-lived: The session remains open for a longer period, potentially throughout a user session or across multiple operations. This allows entities to be reused without repeated database queries. However, it increases the risk of memory leaks, concurrency issues, and stale data. Can be useful in specific cases, such as batch processing or maintaining user state in desktop applications.

> **Note**
>
> Use short-lived sessions by default unless there is a clear need for long-lived sessions. Avoid keeping a session open longer than necessary to reduce memory consumption.
> Flush and commit changes regularly to maintain database consistency.

Be cautious with lazy loading when using detached entities, as it may trigger unexpected database queries outside of an active session. Modern ORM frameworks manage session automatically. Manual session management can be necessary to optimize performances

15.3.3. Transaction

JPA operates in a transactional context: modifications to the database must be executed within a transaction.

A **Transaction** is a sequence of one or more database operations that are executed as a single unit of work. Transactions, following the ACID properties, ensure data integrity by guaranteeing that either all operations within the transaction are successfully completed or none of them take effect. This prevents situations where partial updates could leave the database in an inconsistent state.

Transaction key features:

- Transactions ensure data integrity by grouping multiple operations together.
- A transaction must be explicitly committed to persist changes or rolled back to undo them.
- If an error occurs during a transaction, the system can abort all changes to maintain database consistency.
- Transactions typically operate within an active session, meaning they are tied to the ORM's persistence context.

Transaction operations:

- Begin – Starts a new transaction.
- Commit – Saves all changes made within the transaction permanently.
- Rollback – Reverts all changes made within the transaction.
- Savepoint – Creates a checkpoint within a transaction to allow partial rollbacks.
- Isolation Level Configuration – Defines how transactions interact with each other to manage concurrent access to data.

Transactions must be explicitly started and committed to apply changes. If an error occurs, the transaction can be rolled back to maintain data integrity.

In a multi-user database system, multiple transactions can be executed concurrently. The most common concurrency issues are:

- Dirty Reads – A transaction reads uncommitted changes from another transaction.
- Non-Repeatable Reads – A transaction reads the same row twice but gets different values because another transaction modified it in between.
- Phantom Reads – A transaction executes the same query twice and gets different results because another transaction inserted or deleted rows.

Isolation Levels define how transactions interact with each other and how data consistency is maintained in concurrent operations. A lower isolation level increases performance but increases the risk of incurring in concurrency issues, while a higher isolation level ensures stronger data integrity but may reduce performance.

The possible isolation levels are:

- *Read Uncommitted*

  Transactions can read uncommitted changes from other transactions (dirty reads allowed). Highest concurrency, lowest data consistency.

- *Read Committed*

  A transaction can only read committed changes (no dirty reads). However, non-repeatable reads and phantom reads can still occur.

- *Repeatable Read*

  Prevents dirty reads and non-repeatable reads by ensuring that a row read within a transaction does not change until the transaction completes.

  Phantom reads may still occur.

- *Serializable*

  The strictest level: transactions are fully isolated from each other by locking rows or entire tables. Prevents all concurrency issues but significantly reduces parallelism.

Isolation level depends on the application's performance and consistency requirements. Typically high-performance application use *Read Committed*, while data-critical applications adopt *Serializable*.

Modern ORM frameworks manage transactions automatically Manual transaction management can be necessary to optimize performances

> **Note**
>
> Keep transactions as short as possible to avoid locking resources for too long. Always handle rollback scenarios to prevent data inconsistencies in case of failures. Use proper isolation levels based on the application's concurrency requirements.

`EntityTransaction` is the interface that represents a database transaction in JPA. `EntityManager` allows to retrieve a transaction via the method `getTransaction()` Its implementation is provided by the JPA provider. It provides methods to control transaction flow:

- `begin()`: starts a new transaction,
- `isActive()`: checks if the transaction is currently running,
- `commit()`: saves all changes made since the transaction began,
- `rollback()`: discards all changes if an error occurs.

15.4. JPA Querying

15.4.1. Querying in ORM

ORMs generate SQL automatically based on the structure of the defined entities. Basic queries can be constructed using entity attributes, allowing developers to work at the object level without writing SQL.

Simple queries are fully managed by the ORM (e.g., finding by primary key, inserting, updating, deleting). ORMs optimize queries by dynamically translating object-oriented queries into SQL tailored for the underlying database.

Key advantages:

- Minimizes direct SQL interaction, reducing complexity
- Ensures consistency by enforcing entity relationships and constraints
- Improves code readability and maintainability by abstracting low-level database interactions

When basic queries are not enough, ORMs offer more advanced techniques:

- Criteria API & Query Builders – Programmatic query construction
- Joins and custom queries – Fetching related data efficiently
- Native SQL queries – Direct SQL execution when necessary

Performance considerations:

- Use pagination and indexing to improve query speed
- Avoid N+1 query problems with proper relationship fetching (e.g., JOIN FETCH, batch fetching)
- Cache frequently accessed data

JPA provides multiple ways to query and manipulate data in a relational database:

- JPQL (Java Persistence Query Language): object-oriented query language like SQL but based on JPA entities instead of tables

- Named Queries: precompiled queries defined inside JPA entities for improved performance and reusability
- Criteria API: programmatic approach to dynamically build type-safe queries
- Native Queries: direct execution of raw SQL queries for advanced operations and database-specific optimizations

Each method serves a specific use case:

- JPQL and Named Queries: suitable for structured, static queries
- Criteria API: useful for dynamic queries with runtime conditions
- Native Queries: necessary when JPQL is insufficient or when using database-specific functions

15.4.2. Query interface

`Query` is the JPA interface used to execute queries. Its implementation is provided by the JPA provider. `TypedQuery<T>` is a query subtype, it ensures type safety by specifying the return type.

Created through `EntityManager`, it represents a compiled query ready for execution Supports different query types: JPQL, Named Queries, Native Queries Criteria API follows a different approach to query definition, using a programmatic, type-safe construction mechanism instead of string-based queries

`EntityManager` provides methods to create untyped generic `Query` objects. For these `Query` objects result type is not enforced at compile time, so manual casting may be required:

- `createQuery(String qlString)`: creates a JPQL query
- `createNamedQuery(String name)`: retrieves a predefined Named Query
- `createNativeQuery(String sqlString)`: creates a query using raw SQL
- `createNativeQuery(String sqlString, String resultSetMapping)`: executes a native query using a custom mapping

`EntityManager` also provides methods to create `TypedQuery<T>` objects. This allows the query to return the specified object when executed, ensures type safety, reducing the need for manual casting:

- `createQuery(String qlString, Class<T>)`: creates a typed JPQL query and maps results to entities
- `createNativeQuery(String sqlString, Class<T>)`: creates a typed native query and maps results to entities
- `createNamedQuery(String name, Class<T>)`: retrieves a predefined Named Query and maps results to entities

The `Query` interface provides methods to configure and customize queries before execution. Parameters can be assigned using named or positional parameters. Date and time values require a TemporalType to specify their granularity.

- `setParameter(String name, Object value)`: assigns a value to a named parameter
- `setParameter(int position, Object value)`: assigns a value to a positional parameter
- `setParameter(String name, Date value, TemporalType temporalType)`: assigns a date/time parameter with a specific TemporalType
- `setParameter(int position, Date value, TemporalType temporalType)`: same as above, but for positional parameters

Once a query is configured, it must be executed to retrieve results These are the most common method that execute a query and get the results

- `getResultList()`: retrieves multiple results; return type depends on the query type and operation:
- `getResultStream()`: similar to the previous, but it returns a Stream instead of a List
- `getSingleResult()`: retrieves a single result; return type depends on the query type and operation:
- `executeUpdate()`: executes UPDATE or DELETE queries; returns the number of affected rows

The result type depends on the query type and on the selection it performs

- `TypedQuery<T>` returns objects of the specified type `T`
- Standard `Query` returns `Object` or `Object[]` when selecting multiple fields
- Partial field projections return `Object[]` (each element corresponds to a selected field)
- Aggregates (SUM, COUNT, etc.) return a `Number`, but the actual type depends on the database. Explicit casting is required

The execution method (`getSingleResult()`, `getResultList()`, …) determines if the result is a single item or a collection of iems

Pagination allows limiting the number of results and setting an offset

- `setMaxResults(int maxResults)`: limits the number of results
- `setFirstResult(int startPosition)`: sets the offset for pagination

15.4.3. Java Persistence Query Language (JPQL)

JPQL (Java Persistence Query Language) is the query language defined by JPA to retrieve and manipulate entity data. It is similar to SQL, but operates on JPA entities and their attributes instead of database tables and columns. JPQL queries are translated into SQL by the JPA provider, making them database-independent.

Supports SELECT, UPDATE, DELETE operations, but does not allow INSERT statements. Entities are persisted (i.e. rows are inserted) using `EntityManager.persist()` method.

JPQL enables navigating entity relationships using JOIN, FETCH, and path expressions. Queries can be statically defined (Named Queries) or dynamically created at runtime.

JPQL – Query Structure

Queries must use entity class names and field names, not database-specific identifiers. The alias is required when using conditions or sorting. Joins are based on entity relationships instead of foreign keys, this enables a type-safe navigation through associations

Example:

```
SELECT u                        : <1>
FROM User u                     : <2>
JOIN u.orders o                 : <3>
WHERE o.status = 'SHIPPED'      : <4>
ORDER BY u.name ASC             : <5>
```

1. select all the colums (like User.* in SQL)
2. define an alias since used in sorting
3. the join uses the relation field in the class and defines an alias since uses it in a condition
4. condition on a column (same as in SQL)
5. ordering on a column (same as in SQL)

Supported operators in JPQL:

- Comparison operators: `=`, `<>`, `<`, `>`, `<=`, `>=`
- Logical operators: `AND`, `OR`, `NOT`
- `LIKE` operator: pattern matching (`LIKE '%value%'`)
- `IN` operator: checks if a value is in a list (`IN ('A', 'B', 'C')`)
- `BETWEEN` operator: checks range (`BETWEEN 10 AND 20`)
- `IS NULL` / `IS NOT NULL`

JPQL supports two types of query parameters, which allow values to be dynamically assigned at runtime instead of hardcoding them in the query:

- positional parameters (`?1`, `?2`)

```
Query query = em.createQuery("SELECT u FROM User u WHERE u.age > ?1 AND u.name = ?2");
query.setParameter(1, 30);
query.setParameter(2, "Bob");
```

- named parameters (`:paramName`)

```
TypedQuery<User> query = em.createQuery("SELECT u FROM User u WHERE u.age > :age", User.class
query.setParameter("age", 25);
```

15.4.4. Named queries

Named Queries are predefined, reusable JPQL queries declared at the entity level. They allow better performance because the query is compiled once and cached. Named Queries help centralize query definitions, making the code more maintainable.

They are defined using the `@NamedQuery` annotation in the entity class Useful for static queries that do not change dynamically at runtime Multiple `@NamedQuery` on a single entity should be grouped using `@NamedQueries` to ensures cleaner and more organized code.

In fact some Java versions or JPA implementations do not support multiple standalone `@NamedQuery` annotations.

Named queries example:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "User.findByAge", query = "SELECT u FROM User u WHERE u.age = :age"),
    @NamedQuery(name = "User.findMinors", query = "SELECT u FROM User u WHERE u.age < 18")
})
public class User {
```

```
    @Id private String cf;
    private String name;
    private int age;
}
```

Named Queries are executed using `EntityManager.createNamedQuery()` The returned `Query` or `TypedQuery<T>` object is used to set parameters and execute the query.

```
Query query = em.createNamedQuery("User.findByAge");
query.setParameter("age", 25);
List<User> users = query.getResultList();
TypedQuery<User> query = em.createNamedQuery("User.findMinors", User.class);
List<User> minors = query.getResultList();
```

Native queries allows executing raw SQL directly on the database. They are used when JPQL is insufficient or when leveraging database-specific functions.

Main difference from JPQL:

- Use table and column names instead of entity and field names
- Queries are written in pure SQL, not JPQL
- Created via `createNativeQuery()`, instead of `createQuery()`

Parameter binding remains the same as JPQL, using `setParameter()`. Execution is performed using the same methods as JPQL queries

15.5. Repository Pattern

The Repository Pattern is an architectural pattern that provides an abstraction layer between the application's business logic and the database. It defines a dedicated component responsible for retrieving, persisting, and managing entities, ensuring that data access operations are encapsulated and decoupled from other parts of the system.

Repository key features:

- Acts as an intermediary between the ORM and business logic.
- Encapsulates data access logic within a dedicated component.
- Keeps database operations separate from business logic.
- Provides a structured interface for querying and modifying entities, improving code maintainability.
- Centralizes data access in one place, reducing code duplication and enforcing consistency.
- Supports standard data access patterns, including CRUD operations, query methods, and custom queries.
- Improves testability by abstracting persistence, making it easier to mock DB interactions or repositories entirely.

Repositories provide a standard set of methods to manage entities in the database. The four primary operations (**CRUD**)in a repository are:

- Create – Inserts a new entity into the database.
- Read – Retrieves entities by ID or search criteria.
- Update – Modifies an existing entity and saves changes.
- Delete – Removes an entity from the database.

Using a repository for CRUD operations ensures a consistent API across the application.

Repositories leverage ORM capabilities to query the database using entity attributes. Query methods allow filtering and retrieving entities based on their attributes, reducing the need to write explicit SQL.

Queries can be composed dynamically, combining multiple conditions within repository methods:

- `findByNameAndEmail (name, email)`: Finds an entity based on its name and email attribute.
- `existsByEmail(String email)`: Checks if an entity with a given email exists

Using repository query methods ensures cleaner and more maintainable business logic, keeping data access centralized.

When query methods are not enough, repositories can define custom query methods. Custom query methods are useful for:

- Complex filtering with advanced multiple conditions
- Aggregation queries (e.g., count, sum, averages)
- Joins and nested queries involving multiple entities
- Custom queries can be written using
  - ORM query languages (e.g., JPQL, TypeORM Query Builder)
  - Native SQL queries, for performance optimizations

While custom queries provide flexibility, they should be used carefully to maintain abstraction. If the result structure of a custom query does not match an entity, raw data (tuples) may be returned instead, causing a loss of abstraction

When handling large datasets, repositories provide mechanisms for efficient pagination and sorting.

- Pagination: limits the number of results returned per request to improve performance. Example strategies: limit/offset, cursor-based pagination.
- Sorting: orders results based on specified fields. Example: findAll(Sort.by(“name”).ascending()).

Proper use of pagination and sorting prevents excessive memory consumption and speeds up query execution.

15.6. Hibernate & H2

JPA is designed to be abstract, it requires:

- a JPA provider, implementing all the methods and managing the persistence unit,
- a Database, that will contain the data.

In the course we use Hibernate and H2 respectively for the two roles.

15.6.1. Hibernate

Hibernate is an Object-Relational Mapping (ORM) framework for Java that simplifies database interactions by allowing developers to work with Java objects instead of writing SQL queries manually.

It fully implements the Jakarta Persistence API (JPA), meaning it can function as a pure JPA provider, following all JPA specifications without requiring additional features. However, Hibernate is more than just a JPA provider. When integrated into higher-level frameworks such as Spring or Jakarta EE, it extends JPA with advanced capabilities such as caching, batch processing, and more efficient query execution Hibernate as JPA Provider.

When used as a pure JPA provider, Hibernate strictly adheres to the JPA specification:

- uses JPA interfaces, such as `EntityManagerFactory` and `EntityManager`, to handle persistence.
- manages transactions through `EntityTransaction`
- provides its implementation of all the interfaces defined by JPA, making them available to work with.
- supports JPQL and Criteria API as query languages
- reads standard `persistence.xml` configuration, allowing easy portability across different JPA providers

When integrated with higher-level frameworks like Spring or Jakarta EE, Hibernate extends its basic JPA implementation with additional features, including:

- Sessions: a flexible and powerful alternative to EntityManager with automatic transaction control capabilities
- HQL (Hibernate Query Language): a more powerful alternative to JPQL with additional flexibility
- Advanced Caching Mechanisms: first-level cache (always active) and optional second-level cache for optimized performance
- Batch Processing: efficient handling of bulk data operations to improve write performance
- Event Listeners and Interceptors: hooks that allow executing custom logic during entity lifecycle events
- Flexible Fetch Strategies: advanced options for lazy and eager loading to optimize database queries
- NoSQL Support: ability to interact with NoSQL databases

Although Hibernate works as a pure JPA provider, it also provides additional properties that enhance performance and database interaction. Some database-related settings are better optimized using Hibernate properties. Hibernate adds dialect support, which allows it to optimize SQL for different DBMS. Schema generation via Hibernate is faster and more efficient than the standard JPA mechanism.

Key Hibernate-Specific Properties:

- `hibernate.dialect`: defines the database type, allowing Hibernate to generate optimized SQL
- `hibernate.hbm2ddl.auto`: handles schema generation with more flexibility than JPA's standard option
- `hibernate.show_sql`: enables logging of SQL statements executed by Hibernate
- `hibernate.format_sql`: formats SQL output for better readability Caching

Hibernate automatically enables first-level caching, even when used as a JPA provider. Entities retrieved within the same transaction are stored in the persistence context, avoiding redundant queries.

Hibernate supports lazy loading using proxy objects, even when acting as a JPA provider. When an entity is marked as `FetchType.LAZY`, Hibernate does not load the related entity immediately. Hibernate automatically loads lazy associations when accessed, unless the session is closed. It does not require explicit calls to JOIN FETCH to load lazy associations.

Common issue: `LazyInitializationException`

Occurs if an entity with a lazy-loaded relation is accessed outside an active transaction. Hibernate expects the session to be open when accessing proxies.

15.6.2. H2

H2 is an open-source relational database written in Java, designed to be lightweight, fast, and easy to integrate into Java applications. It requires no installation and supports standard SQL, JDBC, and in-memory operations, making it a flexible solution for both development and testing environments.

H2 is fully compatible with JPA and ORM frameworks, allowing seamless integration with Hibernate. It also includes a web-based console that enables developers to debug queries and inspect the database structure efficiently.

This database supports three different execution modes:

- In-Memory Mode

  The database exists only in RAM and is lost when the application stops. Ideal for unit tests and temporary data storage

  JDBC URL: `jdbc:h2:mem:testdb`

- Embedded Mode

  The database is stored in a local file accessible only from the same Java process. Suitable for small standalone applications.

  JDBC URL: `jdbc:h2:file:./data/mydb`

- Server Mode

  H2 runs as a separate process, allowing multiple clients to connect. Can be accessed remotely via TCP or Web API.

  Useful when multiple applications need to share the same database

  JDBC URL: `jdbc:h2:tcp://localhost/~/mydb`

The H2 database includes a built-in web-based management interface. It provides access to the database for executing queries, modifying data, and managing schemas. Works similarly to traditional database workbenches but runs in a web browser. Useful for debugging, testing, and database administration

To start the console manually:

```
java -jar h2.jar
```

Alternatively is is possible to programmatically start the console server in Java:

```
Server.createWebServer("-web").start();
```

The console runs by default on port 8082: `http://localhost:8082`

There are a few basic properties that must be configure in Hibernate to operate with an H2 databse: These properties must be set in `persistence.xml`:

Database Connection

Driver and Dialect

Schema Management Controls automatic table creation and migration SQL Logging Enables logging of executed queries for debugging and performance analysis Configuration

- Database Connection: defines how the application connects to H2

  ```
  jakarta.persistence.jdbc.driver = org.h2.Driver
  jakarta.persistence.jdbc.url = jdbc:h2:mem:testdb
  jakarta.persistence.jdbc.user = sa (default value)
  jakarta.persistence.jdbc.password = (empty by default)
  ```

  The JDBC URL determines whether the database runs in memory, as a file, or as a server. The JDBC driver must be specified to allow the application to interact with H2.

- Dialect and Schema Management (implemented by hibernate)

  ```
  hibernate.dialect = org.hibernate.dialect.H2Dialect
  hibernate.hbm2ddl.auto = update
  ```

  The SQL dialect helps Hibernate generate efficient database-specific queries.

  The property `hibernate.hbm2ddl.auto` has a set of available values

  - `none`: no automatic schema generation. The database schema must be created manually.
  - `validate`: verifies that the database schema matches the entity definitions but does not create or modify tables. It throws an error if the schema is incorrect
  - `update`: updates the schema without dropping existing tables, it adds missing columns or constraints but does not remove them.
  - `create`: drops existing tables and creates the schema from scratch every time the application starts.
  - `create-drop`: creates the schema from scratch every time the application starts drops it when the session factory is closed.

- SQL Logging (implemented by hibernate)

  ```
  hibernate.show_sql = true
  hibernate.format_sql = true
  ```

15.7. Implementation of an ORM-based application

In this section, we will implement a simple ORM-based application using Hibernate and H2 step-by-step

1. Setting up the environment
2. Persistence configuration
3. Handling EntityManager
4. Defining entities
5. Implementing repositories
6. Implementing a main executor for our application

15.7.1. Dependencies import

First step is to define the `pom.xml` file of the application, to import the necessary dependencies.

- jakarta.persistence-api: JPA API for ORM
- hibernate-core: Hibernate as the JPA provider
- h2: the database

In the `pom.xml`:

```
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>6.4.0.Final</version>
        </dependency>
        <dependency>
            <groupId>jakarta.persistence</groupId>
            <artifactId>jakarta.persistence-api</artifactId>
            <version>3.1.0</version>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>2.1.214</version>
        </dependency>
```

15.7.2. Persistence configuration

In persistence.xml we must set up

- persistence unit
- JPA provider (Hibernate)
- database connection
- driver, URL and access credentials
- sql dialect

- automatic schema generation
- logging and debugging features

The `persistence.xml` must be placed in the `resources/META-INF` folder.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
    <persistence-unit name="flightPU">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="jakarta.persistence.jdbc.url"
                   value="jdbc:h2:file:./data/flightdb"/>
            <property name="jakarta.persistence.jdbc.user" value="sa"/>
            <property name="jakarta.persistence.jdbc.password" value=""/>
            <property name="hibernate.dialect"
                   value="org.hibernate.dialect.H2Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

15.7.3. EntityManager handling

To avoid creating multiple EntityManagerFactory instances, we use a singleton utility class `JPAUtil`. The advantages is that it avoids redundant factory instances since `EntityManagerFactory` is expensive to create.

This class must offer some core features - Ensure proper resource management Providing a `close()` method to shut down Hibernate properly when the - application ends its lifecycle - Encapsulate `EntityManagerFactory` The factory is not exposed directly, but a controlled method provides access to EntityManager Allows application logic to interact only through `EntityManager`, ensuring proper lifecycle management

```
public class JPAUtil {
    private static final EntityManagerFactory emf =
        Persistence.
        createEntityManagerFactory("flightPU");

    public static EntityManager getEntityManager() {
        return emf.createEntityManager();
    }
    public static void close() {
        if (emf.isOpen()) {
            emf.close();
        }
```

```
    }
}
```

15.7.4. Entity definition

In this step we must create the data model of our application:

- Entities will be defined using JPA annotations: how Java objects and their attributes are mapped to database tables and columns.
- Relationships: how Java objects are related and database data are linked across tables

15.7.4.1. Entities

```
@Entity @Table(name = "airplanes")
public class Airplane {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(nullable = false)
    private String model;
    @Column(nullable = false)
    private String airline;

    private int capacity;
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private AirplaneStatus status;
}
```

15.7.4.2. OneToOne relationships

A One-to-One relationship links two entities where each instance of one entity is associated with exactly one instance of another entity By default, One-to-One relationships are `FetchType.EAGER` Owner side It contains the `@JoinColumn` annotation because it holds the foreign key Hibernate considers this entity responsible for managing the relationship Inverse side It uses `mappedBy` to reference the relationship without creating an extra foreign key column It defines cascading and orphanRemoval behavior When querying the inverse side, Hibernate will JOIN the owner table to retrieve the associated object OneToOne relationships

Owner side:

```
@Entity
@Table(name = "passports")
public class Passport {
    @Id
    @Size(min = 8, max = 8)
    private String number;

    @Column(nullable = false)
    private String nationality;

    @OneToOne
    @JoinColumn(name = "person_id",
                nullable = false)
    private Person person;
}
```

Inverse side:

```
@Entity
@Table(name = "people")
public abstract class Person {
    // ...
    @OneToOne(mappedBy = "person",
              cascade = CascadeType.ALL,
              orphanRemoval = true)
    private Passport passport;
}
```

15.7.4.3. OneToMany/ManyToOne relationships

A One-to-Many (`@OneToMany`) means that one entity is related to multiple instances of another entity. A Many-to-One (`@ManyToOne`) means that multiple instances of one entity reference a single instance of another entity. These two annotations represent the same database relationship, just seen from different perspectives.

`@ManyToOne` - Owner Side This side contains the foreign key column in the database This is where the relationship is actually managed Default FetchType.EAGER

```
@Entity
@Table(name = "flights")
public class Flight {
    //...
    @ManyToOne
    @JoinColumn(name="airplane_id",
                nullable=false)
    private Airplane airplane;
}
```

`@OneToMany` - Inverse Side Uses mappedBy to reference the owner Does not create an additional foreign key column Default FetchType.LAZY OneToMany/ManyToOne relationships

```
@Entity
@Table(name = "airplanes")
public class Airplane {
   // ...
    @OneToMany(mappedBy = "airplane")
    private Set<Flight> flights;
}
```

15.7.4.4. ManyToMany relationships

A Many-to-Many relationship means that multiple instances of one entity are associated with multiple instances of another entity This relationship is represented by a join table, which stores the associations between the two entities JPA automatically creates the join table with two foreign keys Default FetchType: `LAZY` (on both sides)

Owner Side

```
@Entity @Table(name = "flights")
public class Flight {
    // ...
    @ManyToMany
    @JoinTable(
        name = "flight_person",
        joinColumns = @JoinColumn(name = "flight_id"),
        inverseJoinColumns = @JoinColumn(name = "person_id")
    )
    private Set<Person> passengers = new HashSet<>();
}
```

The entity that defines the `@JoinTable` annotation owns the relationship. It explicitly specifies the join table name and the foreign key columns.

Inverse side

```
@Entity
@Table(name = "people")
public class Person {
    //...
    @ManyToMany(mappedBy = "passengers")
    private Set<Flight> flights = new HashSet<>();
}
```

Inverse side uses `mappedBy` to reference the owner side. Does not create the join table, as it is already managed by the owner ManyToMany relationships.

15.7.5. Repository

JPA does not define a repository concept, and neither does Hibernate when used as a pure JPA provider. This means that repository classes must be implemented manually to handle entity persistence.

The main purpose of a repository is to limit direct access to `EntityManager`, ensuring a structured and maintainable approach. Instead of passing `EntityManager` throughout the application, repositories encapsulate database operations, exposing only controlled methods for data access.

15.7.5.1. Generic Repository

Since JPA does not define a repository concept, a Generic Repository is a way to standardize common database operations. A `GenericRepository` class defines reusable methods using Java Generics, making it adaptable to different entity types. It allows to centralize basic CRUD operations for any entity type, avoiding code duplication. Create, update, and delete operations can be fully generalized, as they follow the same persistence logic for any entity. The only generic read operations, independent of entity-specific condition, are search by id and unconditioned search. This approach ensures that basic persistence logic is implemented once, and specific repositories can extend it for more advanced queries.

```
public class GenericRepository<T, ID> {
    private final Class<T> entityClass;
    protected GenericRepository(Class<T> entityClass) {
        this.entityClass = entityClass;
    }
    public void update(T entity) {
        EntityManager em = JPAUtil.getEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        em.merge(entity);
        tx.commit();
        em.close();
    }
    public void delete(ID id) {
        EntityManager em = JPAUtil.getEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        T entity = em.find(entityClass, id);
        if (entity != null) {
            em.remove(entity); }
        tx.commit();
        em.close();
    }
    public void save(T entity) {
        EntityManager em = JPAUtil.getEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
```

```
        em.persist(entity);
        tx.commit();
        em.close();
    }
    public T findById(ID id) {
        EntityManager em = JPAUtil.getEntityManager();
        T entity = em.find(entityClass, id);
        em.close();
        return entity;
    }
    public List<T> findAll() {
        EntityManager em = JPAUtil.getEntityManager();
        List<T> result = em.createQuery("SELECT e FROM "
                          + entityClass.getSimpleName()
                          + " e", entityClass)
                .getResultList();
        em.close();
        return result;
    }
}
```

15.7.5.2. Typed Repository

While a Generic Repository provides common CRUD operations for any entity, a Typed Repository is a specialized implementation for a specific entity type.

It extends the Generic Repository and allows defining custom queries tailored to the entity's attributes and business logic. A Typed Repository enables more complex data retrieval using JPQL, Criteria API, or native SQL queries, without exposing raw EntityManager operations in the application logic. This approach ensures a structured separation of concerns, keeping generic persistence logic in the Generic Repository and custom queries in the Typed Repository.

```
public class FlightRepository extends GenericRepository<Flight, Long> {
    public FlightRepository() {
        super(Flight.class);
    }
    public List<Flight> findByAirplaneModel(String model) {
        EntityManager em = JPAUtil.getEntityManager();
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Flight> query = cb.createQuery(Flight.class);
        Root<Flight> root = query.from(Flight.class);
        Join<Flight, Airplane> airplaneJoin = root.join("airplane");
        query.select(root)
        .where(cb.equal(airplaneJoin.get("model"), model));
        List<Flight> flights = em.createQuery(query).getResultList();
        em.close();
        return flights;
```

```
    }
}
```

15.7.6. Main executor

The Main Executor serves as the entry point for the application, handling entity persistence and data manipulation in the database. It is necessary to explicitly close `EntityManagerFactory` when the application shuts down. A shutdown hook is registered to ensure that Hibernate properly releases resources at the end of execution. Note that in frameworks like Spring or Jakarta EE, this process is managed automatically it must be handled manually.

```
public class DatabaseInitializer {

    private static final GenericRepository<Airplane, Long> airplaneRepository = new GenericRepo
    private static final FlightRepository flightRepository = new FlightRepository();

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(JPAUtil::close));
        DatabaseInitializer.initialize();
    }

    public static void initialize() {
        Airplane airplane1 = createAirplane("Boeing 737", "Airways", 180, AirplaneStatus.IN_SER
        Airplane airplane2 = createAirplane("Airbus A320", "Sky Airlines", 160, AirplaneStatus.
        airplaneRepository.save(airplane1);
        airplaneRepository.save(airplane2);

        Flight flight1 = createFlight("FL1234", "2024-04-01T10:00", "2024-04-01T13:00", airplan
        Flight flight2 = createFlight("FL5678", "2024-04-02T12:00", "2024-04-02T16:00", airplan

        flightRepository.save(flight1);
        flightRepository.save(flight2);
    }
}
```

15.8. Testing JPA Applications

To test an application using an ORM:

- Use a PU dedicated to testing using drop-and-create policy
- Each test should start with well defined database contents
- Cleanup is required to avoid poisoning successive test cases
- Use different data (especially keys) in different tests Tests may be executed in parallel so they should not interfere in insert, delete, and updated operations.

Entities are simple data holders, representing the structure of the database

They do not contain business logic, only fields, relationships, and metadata (e.g., constraints)

Testing them is not necessary, as their correctness is enforced by

- The ORM itself, which ensures mappings are correctly applied
- The database schema, which enforces constraints (e.g., foreign keys, uniqueness)
- Other application tests, where entities are implicitly used and validated

The persistence unit in test can be a in-memory db to reduce test time. The database must be recreate often, on every test session. Enable logging to make debugging easier.

```
<property name="jakarta.persistence.jdbc.url"
  value="jdbc:h2:mem:testuniversitydb;DB_CLOSE_DELAY=-1"/>
<property name="jakarta.persistence.schema-generation.database.action" value="drop-and-create"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
```

A typical set-up and tear-down of tests when tests do no write to db

```
    @BeforeAll
    static void populateDb() {
        JPAUtil.setTestMode();
//…
    }

    @AfterAll
    static void cleanDb() {
        JPAUtil.close();
    }
```

While, in a settings where test cases need to write to the db, it must be reset before each test to avoid interference.

```
    @BeforeEach
    static void populateDb() {
        JPAUtil.setTestMode();
//…
    }

    @AfterEach
    static void cleanDb() {
        JPAUtil.close();
    }
```

This latter configuration is much more expensive in terms of computation and time, so it should be adopted when strictly needed.

References

"7-bit coded character set." 1991. Standard. ECMA-6; ECMA. https://ecma-international.org/wp-content/uploads/ECMA-6_6th_edition_december_1991.pdf.

"8-bit single-byte coded graphic character sets - Latin alphabets No. 1 to No. 4." 1986. Standard. ECMA-94; ECMA. https://ecma-international.org/wp-content/uploads/ECMA-94_2nd_edition_june_1986.pdf.

Booch, Grady. 1993. *Object-Oriented Analysis and Design with Applications*. 2nd ed. Benjamin Cummings.

Dijkstra, Edsger W. 1968. "Letters to the Editor: Go to Statement Considered Harmful." *Communications of the ACM*, no. 3, 11: 147–48. https://dl.acm.org/doi/10.1145/362929.362947.

Driessen, Vincent. 2010. "A Successful Git Branching Model." 2010. https://nvie.com/posts/a-successful-git-branching-model/.

Fowler, Martin. 2024. "Continuous Integration." 2024. https://martinfowler.com/articles/continuousIntegration.html.

"Guide to the Software Engineering Body of Knowledge." 2024. IEEE Computer Society. https://ieeecs-media.computer.org/media/education/swebok/swebok-v4.pdf.

IEEE Standard for Configuration Management in Systems and Software Engineering. 2012. New York, NY, USA: Institute of Electrical; Electronics Engineers (IEEE).

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Liskov, Barbara H., and Jeannette M. Wing. 1994. "A Behavioral Notion of Subtyping." *ACM Trans. Program. Lang. Syst.* 16 (6): 1811–41. https://doi.org/10.1145/197320.197383.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. Prentice-Hall.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. 1994. *Object-Oriented Modeling and Design*. Prentice Hall.

Selic, Bran, G. Gullekson, and P. T. Ward. 1994. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc.

Shore, Jim. 2004. "Fail Fast." *IEEE Software* 21 (5): 21–25. https://martinfowler.com/ieeeSoftware/failFast.pdf.

"Unicode Specification." n.d. http://www.unicode.org/versions/latest/.