

## Java (basic concepts)



Version 2 - June 2008

## Control statements

- Same as C
  - ♦ if-else,
  - ♦ switch,
  - ♦ while,
  - ♦ do-while,
  - ♦ for,
  - ♦ break,
  - ♦ continue



Politecnico di Torino

4

## Comments

- C-style comments (multi-lines)

```
/* this comment is so long
   that it needs two lines */
```
- Comments on a single line

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



Politecnico di Torino

2

## Boolean

- Java has an explicit type (boolean) to represent logic values (**true**, **false**)
- Conditional constructs evaluate boolean conditions
  - ♦ **Note well** – It's not possible to evaluate this condition

```
int x = 7; if(x){...} //NO
```
  - ♦ Use relational operators

```
if (x != 0)
```



Politecnico di Torino

5

## Code blocks and Scope

- Java code blocks are the same as in C language
- 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 block code

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



Politecnico di Torino

3

## Passing parameters

- Parameters are always passed **by value**
- ...they can be primitive types or object **references**
- **Note well**: only the object reference is copied not the value of the object



Politecnico di Torino

6

## Constants

- The **final** modifier

```
final float PI = 3.14;
```

```
PI = 16.0; // ERROR, no changes
final int SIZE; // ERROR, init missing
```

- Use uppercases (coding conventions)

## Primitive type

- Defined in the language:
  - ♦ int, double, boolean
- Instance declaration:
  - ♦ Declares instance name `int i;`
  - ♦ Declares the type `0`
  - ♦ Allocates memory space for the reference

## Elements in a OO program

Structural elements  
(types)  
(compile time)

- Class
- Primitive type

Dynamic elements  
(data)  
(run time)

- Reference
- Variable

## Class

- Defined by developer (eg, Exam) or by the Java environment (eg, String)
- the following declaration

```
Exam e; e null
```

- ...allocates memory space for the *reference* ("pointer")
- ...and *sometimes* it initializes it with **null** by default
- Allocation and initialization of the *object* value are made later by its constructor

```
e = new Exam(); e 0Xffe1
```

Object  
Exam

## Classes and primitive types

- |   |                   |  |
|---|-------------------|--|
| <ul style="list-style-type: none"> <li>▪ Class</li> </ul> <pre>class Exam {}</pre>                                | <p>descriptor</p> | <ul style="list-style-type: none"> <li>▪ type primitive</li> </ul> <pre>int, char, float</pre>   |
| <ul style="list-style-type: none"> <li>▪ Variable of type reference</li> </ul> <pre>Exam e; e = new Exam();</pre> | <p>instance</p>   | <ul style="list-style-type: none"> <li>▪ Variable of type primitive</li> </ul> <pre>int i;</pre> |

## Primitive types

## Primitive types

- Have a unique dimension and encoding
  - ♦ Representation is platform-independent

type	Dimension	Encoding
<code>boolean</code>	1 bit	-
<code>char</code>	16 bits	Unicode
<code>byte</code>	8 bits	Signed integer 2C
<code>short</code>	16 bits	Signed integer 2C
<code>int</code>	32 bits	Signed integer 2C
<code>long</code>	64 bits	Signed integer 2C
<code>float</code>	32 bits	IEEE 754 sp
<code>double</code>	64 bits	IEEE 754 dp
<code>void</code>	-	-

## Logical operators

- Logical operators follows C syntax:  
`&& || ! ^`
- **Note well:** Logical operators work **ONLY** on booleans
  - ♦ Type `int` is NOT considered a boolean value like in C
  - ♦ Relational operators work with boolean values

## Constants

- Constants of type `int`, `float`, `char`, strings follow C syntax
  - ♦ `123 256789L 0xff34 123.75 0.12375e+3`
  - ♦ `'a' '%' '\n' "prova" "prova\n"`
- Boolean constants (do not exist in C) are
  - ♦ `true, false`

## Classes

## Operators (integer and floating-point)

- Operators follow C syntax:
  - ♦ arithmetical `+ - * / %`
  - ♦ relational `== != > < >= <=`
  - ♦ bitwise (int) `& | ^ << >> ~`
  - ♦ Assignment `= += -= *= /= %= &=`  
`|= ^=`
  - ♦ Increment `++ --`
- Chars are considered like integers (e.g. `switch`)

## Class

- Object descriptor
- It consists of attributes and methods

## Class – definition

```
class Car {
    String color;
    String brand;
    boolean turnedOn;
    void turnOn() {
        turnedOn = true;
    }
    void paint (String newCol) {
        color = newCol;
    }
    void printState () {
        System.out.println("Car " + brand + " " + color);
        System.out.println("the engine is"
            +(turnedOn?"on":"off"));
    }
}
```



## Overloading

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

## Methods

- Methods are the messages that an object can accept
  - ♦ `turnOn`
  - ♦ `paint`
  - ♦ `printState`
- Method may have parameters
  - ♦ `paint("Red")`

## Objects

- An object is identified by:
  - ♦ Its class, which defines its structure (attributes and methods)
  - ♦ Its **state** (attributes values)
  - ♦ An **internal unique identifier**
- Zero, one or more reference can point to the same object

## Overloading

- In a Class there may be different methods with the same name
- But they have a different **signature**
- A signature is made by:
  - ♦ Method name
  - ♦ Ordered list of parameters types
- the method whose parameters types list matches, is then executed

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

## Objects

```
class Car {
    String color;
    void paint(){
        color = "white";
    }
    void paint(String newCol) {
        color = newCol;
    }
}
Car a1, a2;
a1 = new Car();
a1.paint("green");
a2 = new Car();
```

## Objects and references

```
Car a1, a2; // a1 and a2 are uninitialized
a1 = new Car();
a1.paint("yellow");// Car "yellow" generated,
// a1 is a reference pointing to "yellow"
a2 = a1; //now two references point to "yellow"
a2 = null; // a reference points to "yellow"
a1 = null;// no more references point to "yellow"
// Object exists but it is no more reachable
// then it will be freed by
// the garbage collector
```

- Note Well: a reference IS NOT an object

SoftEng

Politecnico di Torino

25

## Heap

- It is a part of the memory used by an executing program...
- ...to store data dynamically created at run-time
- C: malloc, calloc and free
  - ♦ Instances of types in static memory or in heap
- Java: new
  - ♦ Instances (Objects) are always in the heap

SoftEng

Politecnico di Torino

28

## Objects Creation

- Creation of an object is made with the keyword **new**
- It returns a reference to the piece of memory containing the created object

```
Motorcycle m = new Motorcycle();
```

SoftEng

Politecnico di Torino

26

## Constructor

- Constructor method contains operations (initialization of attributes etc.) we want to execute on each object as soon as it is created
- Attributes are always initialized
  - ♦ Attributes are initialized with default values
- If a Constructor is not declared, a default one (with no parameters) is defined
- Overloading of constructors is often used

SoftEng

Politecnico di Torino

29

## The keyword new

- Creates a new instance of the specific Class, and it allocates the necessary memory in the heap
- Calls the **constructor** method of the object (a method without return type and with the same name of the Class)
- Returns a reference to the new object created
- Constructor can have parameters
  - ♦ String s = new String("ABC");

SoftEng

Politecnico di Torino

27

## Constructors with overloading

```
class Window {
    String title;
    String color;
    Window() {
        // creates a window with no title and no color
    }
    Window(String t) {
        // creates a window with title but no color
        ...
        title = t;
    }
    Window(String t, String c) {
        // creates a window with title and color...
        title = t;
        color = c;
    }
}
```

SoftEng

Politecnico di Torino

30

## Destruction of objects

- It is no longer a programmer concern
- See section on Memory management
- Before the object is really destroyed if exists method finalize is invoked:

```
public void finalize()
```

## Caveat (cont'd)

- In such cases **this** is implied
- "this" is a reference to the current object

```
class Book {
    int pages;
    void readPage(int n){...}
    void readAll() {
        for(...)
            readPage(i);
    }
}
```

```
void readAll() {
    for(...)
        this.readPage(i);
}
```

## Methods invocation

- A method is invoked using dotted notation  
`objectReference.Method(parameters)`

- Example:

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

## Access to attributes

- Dotted notation  
`objectReference.attribute`
  - ♦ Reference is used like a normal variable

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

## Caveat

- If a method is invoked within another method of the **same** object
  - ♦ DO NOT need to use dotted notation

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

## Access to attributes

- Methods accessing attributes of the same object do not need using object reference

```
class Car {
    String color;

    void paint(){
        color = "green";
        // color refers to current obj
    }
}
```

## this

- It can be useful in methods to distinguish object attributes from local variables
  - ♦ this represents a reference to the current object

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

## Strings



## Combining dotted notations

- Dotted notations can be combined  
`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 on the screen

## String

- No primitive type to represent string
- String literal is a quoted text
- C
  - ♦ `char s[] = "literal"`
  - ♦ Equivalence between string and char arrays
- Java
  - ♦ `char[] != String`
  - ♦ **String class** in `java.lang` library

## Operations on references

- Only the relational operators `==` and `!=` are defined
  - ♦ Note well: the equality condition is evaluated on the values of the references and NOT on the values of the objects !
  - ♦ The relational operators tell you whether the references points to the same object in memory
- Dotted notation is applicable to object references
- There is **NO** pointer arithmetic

## String and StringBuffer

- class `String` (`java.lang`)
  - ♦ **not modifiable**
- class `StringBuffer` (`java.lang`)
  - ♦ **Modifiable**

```
String s = new String("literal")
StringBuffer sb=new StringBuffer("literal")
```

## Operator +

- It is used to **concatenate** 2 strings  
`"This string" + " is made by two strings"`
- Works also with other types (automatically converted to string)  
`System.out.println("pi = " + 3.14);`  
`System.out.println("x = " + x);`

## StringBuffer

- insert
- append
- delete
- reverse

## String

- `int length()`
  - ♦ returns string length
- `boolean equals(String s)`
  - ♦ compares the values of 2 strings

```
String s1, s2;  
s1 = new String("First string");  
s2 = new String("First string");  
System.out.println(s1);  
System.out.println("Length of s1 = " +  
s1.length());  
if (s1.equals(s2)) // true  
if (s1 == s2) // false
```

## Character

- Utility methods on the kind of char
  - ♦ `isLetter()`, `isDigit()`,  
`isSpaceChar()`
- Utility methods for conversions
  - ♦ `toUpperCase()`, `toLowerCase()`

## String

- `String valueOf(int)`
  - ♦ Converts int in a String - available for all primitive types
- `String toUpperCase()`
- `String toLowerCase()`
- `String substring(int startIndex)`
- `int indexOf(String str)`
  - ♦ Returns the index of the first occurrence of *str*
- `String concat(String str)`
- `int compareTo(String str)`

## Scope and encapsulation

## Example

- Laundry machine, design1
  - ♦ commands:
    - time, temperature, amount of soap
  - ♦ Different values depending if you wash cotton or wool, ....
- Laundry machine, design2
  - ♦ commands:
    - key C for cotton, W for wool, Key D for knitted robes

SoftEng

Politecnico di Torino

49

## Scope and Syntax

- private (applied to attribute or method)
  - ♦ attribute/method visible by instances of the same class
- public
  - ♦ attribute / method visible everywhere
- protected
  - ♦ attribute / method visible by instance of the same class and sub-classes
- Forth type (later)

SoftEng

Politecnico di Torino

52

## Example (cont'd)

- Washing machine, design3
  - ♦ command:
    - Wash!
  - ♦ insert clothes, and the washing machine automatically select the correct program
- Hence, there are different solutions with different level of granularity / abstraction

SoftEng

Politecnico di Torino

50

## Info hiding

```
class Car {
    public String color;
}
Car a = new Car();
a.color = "white"; // ok
```



better

```
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
```

SoftEng

Politecnico di Torino

53

## Motivation

- Modularity = cut-down inter-components interaction
- Info hiding = identifying and delegating responsibilities to components
  - ♦ components = Classes
  - ♦ interaction = read/write attributes
  - ♦ interaction = calling a method
- Heuristics
  - ♦ attributes invisible outside the Class
  - ♦ Visible methods are the ones that can be invoked from outside the Class

SoftEng

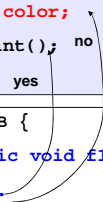
Politecnico di Torino

51

## Info hiding

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

```
class B {
    public void f1(){
        ...
    };
}
```



SoftEng

Politecnico di Torino

54

## Access

	Method in the same class	Method of another class
Private (attribute/method)	yes	no
Public	yes	yes

SoftEng

Politecnico di Torino

55

## Example without getter/setter

```
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;
        // print vote
        System.out.println(e.grade);
        // print student
        System.out.println(e.student.last);
    }
}
```

SoftEng

Politecnico di Torino

58

## Getters and setters

- Methods used to read/write a private attribute
- 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;
}
```

SoftEng

Politecnico di Torino

56

## Example with getter/setter

```
class StudentExample {
    public static void main(String[] args) {
        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();
    }
}
```

SoftEng

Politecnico di Torino

59

## Example without getter/setter

```
public class Student {
    public String first;
    public String last;
    public int id;
    public Student(...){...}
}
public class Exam {
    public int grade;
    public Student student;
    public Exam(...){...}
}
```

SoftEng

Politecnico di Torino

57

## Example with getter/setter

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

SoftEng

Politecnico di Torino

60

## Example with getter/setter

```
public class Exam {
    private int grade;
    private Student student;

    public void print() {
        System.out.println("Student " +
            student.toString() + "got " + grade);
    }

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

## Array declaration

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

```
int[] a;
int a[];
```

- In Java an array is an **Object** and it is **stored in the heap**
- Array declaration allocates memory space for a **reference**, whose default value is null

a null

## Array

## Array creation

- Using the **new** operator...

```
int[] a;
a = new int[10];
String[] s = new String[5];
```

- ...or using **static initialization**, filling the array with values

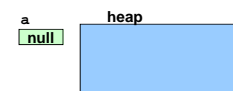
```
int[] primes = {2,3,5,7,11,13};
Person[] p = { new Person("John"),
               new Person("Susan") };
```

## Array

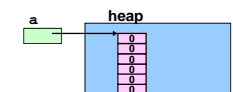
- An array is an **ordered sequence** of variables of the same type which are accessed through an **index**
- Can contain both **primitive types** or **object references** (but no object values)
- Array **dimension** can be defined at run-time, during object creation (cannot change afterwards)

## Example – primitive types

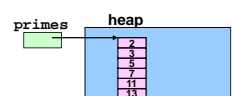
```
int[] a;
```



```
a = new int[6];
```

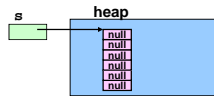


```
int[] primes =
    {2,3,5,7,11,13};
```

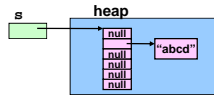


## Example – object references

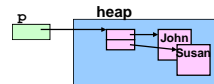
```
String[] s = new
String[6];
```



```
s[1] = new
String("abcd");
```



```
Person[] p =
{new Person("John"),
new Person("Susan")};
```



## For each

- New loop construct:
 

```
for( Type var : set_expression )
```

  - ♦ Notation very compact
  - ♦ *set\_expression* can be
    - either an array
    - a class implementing `Iterable`
  - ♦ The compiler can generate automatically loop with correct indexes
  - ♦ Thus less error prone

## Operations on arrays

- Elements are selected with brackets `[]` (C-like)
  - ♦ But Java makes **bounds checking**
- Array length (number of elements) is given by attribute **length**

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

## For each – example

- Example:
 

```
for(String arg: args){
    //...
}
♦ is equivalent to
for(int i=0; i<args.length;++i){
    String arg= args[i];
    //...
}
```

## Operations on arrays

- An array reference is **not** a pointer to the first element of the array
- It is a pointer to the array **object**
- **Arithmetic on pointers does not exist in Java**

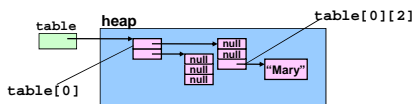
## Homework

- Create an object representing an ordered list of integer numbers (at most 100)
- `print()`
  - ♦ prints current list
- `add(int)` and `add(int[])`
  - ♦ Adds the new number(s) to the list

## Multidimensional array

- Implemented as array of arrays

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



## Tartaglia's triangle

- Write an application printing out the following Tartaglia's triangle

```
1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1  
1 5 10 10 5 1  
1 6 15 20 15 6 1
```

Diagram illustrating the calculation of the value 4 in the 4th row, 3rd column:  $4 = 3 + 1$ . The value 3 is from the cell above-left and 1 is from the cell above-right.

## Rows and columns

- As **rows** are not stored in adjacent positions in memory they can be **easily exchanged**

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

## Package



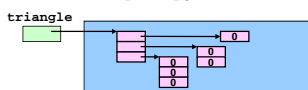
## Rows with different length

- A matrix (bidimensional array) is indeed an array of arrays

```
int[][] triangle = new int[3][]
```



```
for (int i=0; i < triangle.length; i++)  
    triangle[i] = new int[i+1];
```



## Motivation

- Class is a better element of **modularization** than a procedure
- But it is still little
- For the sake of organization, Java provides the package feature

## Package

- A package is a **logic set** of class definitions
- These classes are made of several files, all stored in the **same directory**
- Each package defines a new **scope** (i.e., it puts bounds to visibility of names)
- It's then possible to use **same class names in different package** without name-conflicts

## Creation and usage

- Creation:
  - ♦ Package statement at the beginning of each class file
- Usage:
  - ♦ Import statement at the beginning of class file (where needed)

```
package packageName;
```

```
import packageName.className;
```

```
import java.awt.*;
```

Import single class (class name is in scope)

Import all classes but not the sub packages

## Package name

- A package is identified by a name with a hierarchic structure (*fully qualified name*)
  - ♦ E.g. **java.lang** (String, System, ...)
- Conventions to create unique names
  - ♦ Internet name in reverse order
  - ♦ **it.polito.myPackage**

## Access to a class in a package

- Referring to a method/class of a package

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

- If two packages define a class with the same name, they cannot be both imported
- If you need both classes you have to use one of them with its fully-qualified name:

```
import java.sql.Date;
```

```
Date d1; // java.sql.Date
```

```
java.util.Date d2 = new java.util.Date();
```

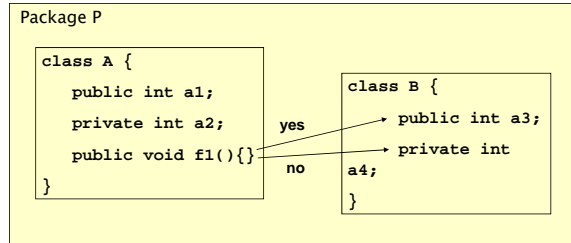
## Example

- java.awt
  - ♦ Window
  - ♦ Button
  - ♦ Menu
- java.awt.event (sub-package)
  - ♦ MouseEvent
  - ♦ KeyEvent

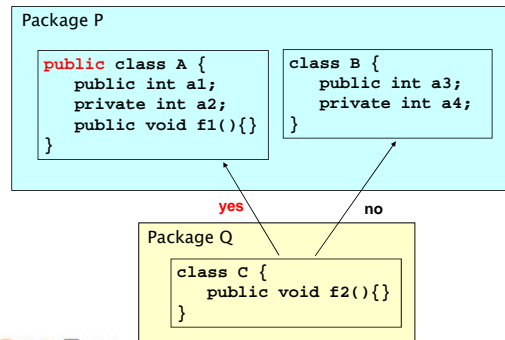
## Package and scope

- Scope rules also apply to packages
- The “interface” of a package is the set of **public classes** contained in the package
- Hints
  - ♦ Consider a package as an entity of modularization
  - ♦ Minimize the number of classes, attributes, methods visible outside the package

## Package visibility



## Multiple packages



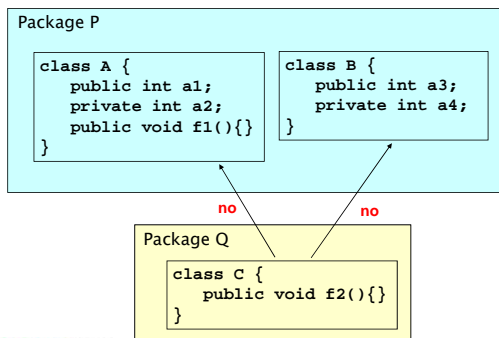
## Visibility w/ multiple packages

- public class A { }
  - ♦ Public methods/attributes of A are visible outside the package
- class B { }
  - ♦ No method/attribute of B is visible outside the package

## Access rules

	Method in the same class	Method of other class in the same package	Method of other class in other package
Private attribute/method	Yes	No	No
Package attribute / method	Yes	Yes	No
Public attribute / method on package class	Yes	Yes	No
Public attribute / method on public class	Yes	Yes	Yes

## Multiple packages



## Static attributes and methods

## Class variables

- Represent properties which are common to all instances of an object
- But they exist even when no object has been instantiated
- They are defined with the **static** modifier
- Access: **ClassName.attribute**

```
class Car {
    static int numberOfWheels = 4;
}

int y = Car.numberOfWheels;
```

## Enum

- Enum can be declared outside or inside a class, but NOT within a method
- Enums are not Strings or ints, but more like a kind of class but constructor can't be invoked directly, conceptually like this:

```
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);
    public Suits (String enumName, int index) {...}
}

SoftEng
Politecnico di Torino
```



## Static methods

- Static methods are not related to any instance
- They are defined with the **static** modifier
- Access: **ClassName.method()**

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

double y = Math.cos(x); // static method
```

## Wrapper classes for built-in types



## Enum

- Defines an enumerative type

```
public enum Suits {
    SPADES, HEARTS, DIAMONDS, CLUBS
}
```
- Variables of enum types can assume only one of the enumerated values

```
Suits card = Suits.HEARTS;
```
- They allow much more strict static checking compared to integer constants (used e.g. in C)

## Motivation

- In an ideal OO world, there are only classes and objects
- For the sake of efficiency, Java use primitive types (int, float, etc.)
- **Wrapper classes** are object versions of the primitive types
- They define **conversion operations** between different types

## Wrapper Classes

Defined in `java.lang` package

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

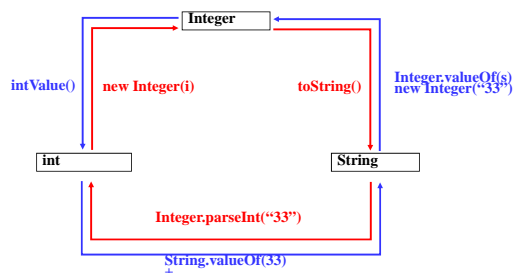
## Autoboxing



- In **Java 5** an automatic conversion between primitive types and wrapper classes (autoboxing) is performed.

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

## Conversions



## Variable arguments



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

`method( Object ... args )`

- The compiler assembles an Object array that can be used to scan the passed parameters

## Example

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

## Variable arguments – example



```
static void plst(String pre, Object...args){  
    System.out.print(pre);  
    for(Object o:args){  
        if(o!=args[0]) System.out.print(", ");  
        System.out.print(o);  
    }  
    System.out.println();  
}  
public static void main(String[] args) {  
    plst("List:", "A", 'b', 123, "ciao");  
}
```

## Memory management



## Types of variables

- **Instance variables** (or fields or attributes)
  - ♦ Stored within objects (in the heap)
- **Local Variables**
  - ♦ Stored in the Stack
- **Static Variables**
  - ♦ Stored in static memory



Politecnico di Torino

106

## Memory types

Depending on the kind of elements they include:

- **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 created in a code block (local variables and method parameters)



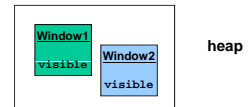
Politecnico di Torino

104

## Instance Variables

- Declared within a Class (attributes)

```
Class Window {  
    boolean visible;  
    ...  
}
```



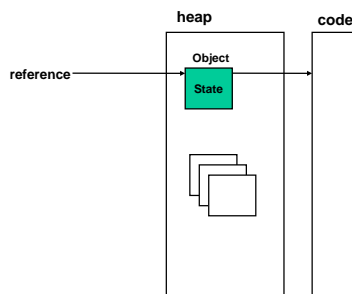
- There is a different copy in each instance of the Class
- Create/initialized whenever a new instance of a Class is created



Politecnico di Torino

107

## Objects are stored in the heap



Politecnico di Torino

105

## Local Variables

- Declared within a method or a code block
- Stored in the stack
- Created at the beginning of the code block where they are declared
- Automatically destroyed at the end of the code block

```
Class Window {  
    ...  
    void resize () {  
        int i;  
        for (i=0; i<5; i++) { ... }  
    } // here i is destroyed  
}
```



Politecnico di Torino

108

## Static Variables

- Declared in classes or methods with **static** modifier

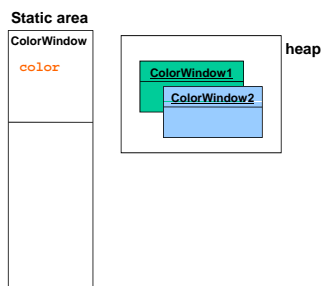
```
Class ColorWindow {  
    static String color;  
    ...  
}
```

- Only ONE copy is stored in static memory
  - ♦ associated to a Class => also called Class variables
- Created/initialized during Class-loading in memory

## Garbage collector

- Is a component of the JVM that has to clean heap memory from 'dead' objects
- After a period of time it analyzes references and objects in memory
- ...and then it deallocates objects with no active references

## Static variables in memory



## Object destruction

- It's not made explicitly but it is made by the JVM when there are no more references to the object
  - ⇒ Programmer must not worry about objects destruction