Object-Oriented Programming

Learning objectives

- Define the object-oriented (OO) paradigm
  - What are objects and classes?
- Understand the differences between procedural approach and OO
  - What is encapsulation?
- Understand the fundamental concepts of OO
  - What are interfaces, messages, and inheritance?
- Appreciate the benefits of OO
  - What are modularity, reuse, and maintainability?

Programming paradigms

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

Languages timeline

<table>
<thead>
<tr>
<th>Procedural languages</th>
<th>Global variables</th>
<th>Data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st gen HLL FORTRAN COBOL ALGOL60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd gen HLL ALGOL68 PASCAL C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3rd gen HLL ADA MODULA2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object-oriented languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMULA67</td>
</tr>
<tr>
<td>SMALLTALK</td>
</tr>
<tr>
<td>C++ EIFFEL ADA9X</td>
</tr>
<tr>
<td>JAVA</td>
</tr>
</tbody>
</table>

‘60 ‘70 ‘80 ‘90
Procedural

```c
int vect[20];
void sort() { /* sort */ }
int search(int n) { /* search */ }
void init() { /* init */ }
// ...
int i;
void main(){
   init();
   sort();
   search(13);
}
```

Problems

- There is no syntactic relationship between:
  - Vectors (int vect[20])
  - Operations on vectors (search, sort, init)

- There is no control over size:
  ```c
  for (i=0; i<=20; i++) vect[i]=0;
  ```

- Initialization
  - Actually performed?

The vector

- It's not possible to consider a vector as a primitive and modular concept
- Data and functions cannot be modularized properly
Procedural – problems

- No constraints on read/write relationships
- External functions can read/write vector’s data

---

Procedural – On the long run

- (All) functions may read/write (all) data
- As time goes by, this leads to a growing number of relationships
- Source code becomes difficult to understand and maintain
  - Problem known as “Spaghetti code”

---

What is OO?

- Procedural Paradigm
  - Program defines data and then calls subprograms acting on data
- OO Paradigm
  - Program creates objects that encapsulate the data and procedures operating on data
- OO is simply a new way of organizing a program
  - Cannot do anything using OO that can’t be done using procedural paradigm

---

Why OO?

- Programs are getting too large to be fully comprehensible by any person
- There is need of a way of managing very-large projects
- Object Oriented paradigm allows:
  - programmers to use large blocks of code
  - without knowing all the picture
- Makes code reuse a real possibility
- Easier maintenance and evolution of code
Why OO?
- Benefits only occur in larger programs
- Analogous to structured programming
  - Programs < 30 lines, spaghetti is as understandable and faster to write than structured
  - Programs > 1000 lines, spaghetti is incomprehensible, probably doesn’t work, not maintainable
- Only programs > 1000 lines benefit from OO really

An engineering approach
- Given a system, with components and relationships among them, we have to:
  - Identify the components
  - Define component interfaces
  - Define how components interact each other through their interfaces
  - Minimize relationships among components

An engineering approach
- Objects introduce an additional abstraction layer
- More complex system can be built

Procedural vs. OO

![Procedural vs. OO diagram]
Object-Oriented approach

- Defines a new component type
  - Object (and class)
  - Data and functions on data are within the same module
  - Allows defining a more precise interface
- Defines a new kind of relationship
  - Message passing
  - Read/write operations are limited to the object scope

Object-Oriented approach

```java
class Vector {

    //data
    private int v[20];

    //interface
    public Vector() {
        for(int i=0; i<20; i++) v[i]=0;
    }
    public sort(){ /*sort*/ }
    public search(int c){ /*search*/ }
 }
```

Object-Oriented approach

- Use of the class Vector:
  - Vector v1 = new Vector();
  - Vector v2 = new Vector();
  - v1.sort();
  - v1.search(22);

Object-Oriented approach

```java
/* The same example in Java */
class Vector {
    private int v[20];
    public Vector() {
        for(int i=0; i<20; i++) v[i]=0;
    }
    public sort() /*sort*/
    public search(int c) /*search*/
 }

/* The same main() in Java */
int main() {  Vector v1 = new Vector();
              Vector v2 = new Vector();
              v1.sort();  v1.search(22);
}```
Class and object

- **Class** (the description of object structure, i.e. *type*):
  - Data  (ATTRIBUTES or FIELDS)
  - Functions (METHODS or OPERATIONS)
  - Creation methods  (CONSTRUCTORS)

- **Object** (class instance)
  - State and identity

---

Example

- Class `car` {
  
  string bodyColor;
  
  void turnOn() {...}
}

- `mikeCar:Object [bodyColor := “blue”]`
- `joeCar:Object [bodyColor := “red”]`
- `maryCar:Object [bodyColor := “blue”]`

---

Class and object

- A class is like a type definition
  - No data 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*
- No limit to the number of objects that can be created from a class
- Each object is independent. Changing one object doesn't change the others

---

UML

```
Student
first
last
studID
print()
```

```
s1
first = Mario
last = Rossi
studID = 1234
print()
```

```
s2
first = Giovanni
last = Verdi
studID = 1237
print()
```

```
class
```

```
objects
```
Message passing
- Objects communicate by message passing
  - Not by procedure call
  - Not by direct access to object’s local data

```
Object v1
v1.sort();
v1.search(22);
```

Message
- A message is a service request
  - search, sort
- A message may have arguments
  - A value or an object name

Examples
- search(21)
- search(joeCar)

Interface
- Set of messages an object can receive
- Any other message is illegal
- The message is mapped to a function within the object
- The object is responsible for the association (message, function)
Interface (simple)

- The interface of an object is simply the subset of methods that other "program parts" are allowed to call
  - Stable

```
Stable Interface
Rest of the Program
```

Encapsulation

```
Object v1
v1.sort();
v1.search(22);
v1.v[0] = 12; // NO
```

- Read/write operations can only be performed by an object on its own data
- Between two objects data are exchanged through message passing

Benefits of encapsulation

- To use an object, the user need only comprehend the interface. No knowledge of the internals are necessary
- Self-contained. Once the interface is defined, the programmer can implement the interface (write the object) without interference of others

Benefits of encapsulation

- Implementation can change at a later date without rewriting any other part of the program (as long as the interface doesn't change)
- Changes in the data mean changing code in one location, rather than code scattered around the program (error prone)
Encapsulation in real life

- PhoneBook
  - Allows user to enter, look up and delete names and phone numbers
  - Implemented using an array
  - Maximum 100 names in the phone book

- PhoneBook object
  - Hidden Data
    - array
  - Interface
    - add, delete, lookUp

Encapsulation in real life

- The PhoneBook object is successful. It is used in hundreds of applications across the company
- It only holds 100 records! It now must be upgraded to hold unlimited number of records
- How do we do so without breaking all the other programs in the company?

Encapsulation in real life

- PhoneBook
  - Allows user to enter, look up and delete names and phone numbers
  - Implemented using an array
  - Maximum 100 names in the phone book

PhoneBook Example

- The interface does not need to change. Thus there is no need to change any of the programs using PhoneBook object
- If this had been programmed in the procedural paradigm, each program that used the phone book would have had a copy of the data array and would have to have been extensively modified to be upgraded
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

Example

- Class Employee{
  string name;
  double wage;
  void incrementWage(){...}
}
- Class Manager extends Employee{
  string managedUnit;
  void changeUnit(){...}
}
- Manager m = new Manager();
  m.incrementWage(); // OK, inherited

Overriding

- Class Vector{
  int vect[20];
  void add(int x) {...}
}
- Class OrderedVector extends Vector{
  void add(int x){...}
}

Why 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
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

Inheritance in real Life

- A new design created by the modification of an already existing design
  - The new design consists of only the changes or additions from the base design
- CoolPhoneBook inherits PhoneBook
  - Add mail address and cell number

Example of inheritance tree

Inheritance and polymorphism

- Class Employee{
  private string name;
  public void print(){
    System.out.println(name);
  }
}
- Class Manager extends Employee{
  private string managedUnit;
  public void print(){ //overrides
    System.out.println(name); //un-optimized!
    System.out.println(managedUnit);
  }
}
Inheritance and polymorphism

```java
void printEmployee(Employee e) {
    e.print();
}

Employee e1 = new Employee();
Employee e2 = new Manager(); // ok, is a
printEmployee(e1); // name
printEmployee(e2); // name and unit
```

Wrap-up session

- Class
  - Data structure (most likely private)
  - Private methods
  - Public interface
- Objects are class instances
  - State
  - Identity

Wrap-up session

- The key role of interfaces
- Objects communicate by means of messages
- Each object manages its own state (data access)

Wrap-up session

- Abstraction
  - The ability for a program to ignore some aspects of the information it’s manipulating, i.e. the ability to focus on the essential
  - Each object in the system serves as a model can perform work, report on and change its state, and “communicate” with other objects in the system, without revealing how these features are implemented
- Example
  - Vector of integers implemented as an array or a linked list
ENCAPSULATION

- Also called information hiding
- Ensures that objects cannot change the internal state of other objects in unexpected ways
- Only the object's own methods are allowed to access its state
- Each type of object exposes an interface to other objects that specifies how other objects may interact with it

Do not brake it, never ever! ...Unless you know what you are doing!
- Loosens up relationships among components

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)

BENEFITS OF OO

- Modularity (no spaghetti code)
- Maintainability
- Reusability