

Programming personal reference
Concepts in a nutshell

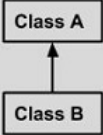
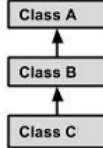
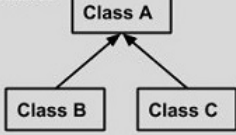
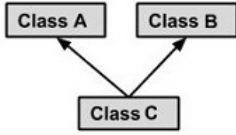
Nicolò Fornari

October 4, 2016

Chapter 1

OOP

1.1 Inheritance

Single Inheritance  <pre>graph BT; B[Class B] --> A[Class A]</pre>	<pre>public class A { } public class B extends A { }</pre>
Multi Level Inheritance  <pre>graph BT; C[Class C] --> B[Class B]; B --> A[Class A]</pre>	<pre>public class A { } public class B extends A { } public class C extends B { }</pre>
Hierarchical Inheritance  <pre>graph BT; B[Class B] --> A[Class A]; C[Class C] --> A</pre>	<pre>public class A { } public class B extends A { } public class C extends A { }</pre>
Multiple Inheritance  <pre>graph BT; C[Class C] --> A[Class A]; C --> B[Class B]</pre>	<pre>public class A { } public class B { } public class C extends A,B { } // Java does not support mutiple Inheritance</pre>

With classes extending other classes when I call a method the nearest one is executed. The concept is similar with variable scope.
Every class inherits from Object.

Example

```
public class MySlider extends javafx.scene.control.Slider {  
  
    public MySlider() {  
        super(0,255,0);  
  
        setShowTickLabels(true);  
        setShowTickMarks(true);  
        setMajorTickUnit(50);  
        setMinorTickCount(5);  
        setBlockIncrement(10);  
    }  
}
```

1.2 Polimorphism

Polymorphism is the ability of an object to take on many forms. Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

```
public class Main {  
    public static void main(String[] args) {  
        Stack s;  
        if (args[0] != "")  
            s=new Coda();  
        else  
            s = new Pila();  
        s.inserisci(1);  
        s.inserisci(2);  
        s.inserisci(3);  
        for (int k=0;k<=4;k++){  
            int v=s.estrai();  
            System.out.println(v);  
        }  
    }  
}
```

1.3 Abstraction

in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

- If a class has an abstract method then it has to be declared abstract
- An abstract class can not be instantiated
- To use an abstract class you have to inherit it from another class and provide the implementation for the declared methods

```
import java.util.*;
public abstract class Stack extends LinkedList {
    public void inserisci(int x) {
        Number n = new Number(x);
        this.add(n);
    }

    abstract public int estrai();
}
```

```
import java.util.*;

class Pila extends Stack {
    @Override
    public int estrai() {
        Number x = null;
        Iterator iter = this.iterator();
        while (iter.hasNext()) {
            x = (Number) iter.next();
        }
        if (x == null) {
            System.out.println("Pila vuota");
            System.exit(1);
        }
        iter.remove();
        return x.getInt();
    }
}
```

1.4 Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class (called setter and getter).

Implementation:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values

Benefits

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

1.5 Interfaces

An interface is a collection of abstract methods.

Interface	Abstract class
Only non implemented methods	It can have implemented methods
No variables	-
A class can implement multiple interfaces	No multiple hierarchy

```
public class drawCircle implements EventHandler {

    Color color;
    Canvas canv;
    int r = 50;

    public drawCircle(Color color,Canvas canv) {
        this.color = color;
        this.canv = canv;
    }

    @Override
    public void handle(Event t) {
        double w = canv.getWidth();
        double h = canv.getHeight();

        int x = (int) (Math.random()*w);
        int y = (int) (Math.random()*h);

        canv.getGraphicsContext2D().setFill(color);
        canv.getGraphicsContext2D().fillOval(x,y,r,r);
    }
}
```


1.6 Packages

Packages in Java are used to control access, making usage of classes and interfaces easier and to prevent naming conflicts.

It is a good practice to define your own package: in this way it is easy to determine which classes and interfaces are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages.

1.7 Modifiers

Access modifiers

- (default) visible to the package
- private: visible to the class only
- protected: visible to the package and all subclasses
- public: visible to the world

Non access

- static
Variables and methods associated to a class rather than to an object are static. Static variables are used as shared variables among instances. Static methods can be called without any instance of the class.
- final
 1. final variables are constant
 2. final methods can not be overwritten
 3. final classes can not be subclassed
- abstract
 1. abstract classes must be subclassed and can not be instantiated
 2. abstract methods must be overwritten

1.8 Collections

A collection is an object that groups multiple elements in a single entity. Since a collection takes as input Objects, if I want to store primitive variables (eg. integers) I need a wrapper.

```
public class Mazzo {  
  
    List<Carta> Deck;  
    int[] valori = {1,2,3,4,5,6,7,8,9,10};  
    String[] semi = {"Denari", "Spade", "Bastoni", "Coppe"};  
  
    public Mazzo() {  
        Deck = new ArrayList<>();  
        for (int i=0; i<4; i++) {  
            for (int j=0; j<10; j++) {  
                Carta newCard = new Carta(semi[i], valori[j]);  
                Deck.add(newCard);  
            }  
        }  
        Collections.shuffle(Deck);  
    }  
}
```

The Collections class provides three constants:

- *Collections.EMPTY_SET*
- *Collections.EMPTY_LIST*
- *Collections.EMPTY_MAP*

The main use of these constants is as input to methods that take a Collection of values, when you don't want to provide any values at all.

1.9 Object ordering - comparable

A List *l* may be sorted as follows: `Collections.sort(l);`

If the list consists of String elements, it will be sorted into lexicographic (alphabetical) order. If it consists of Date elements, it will be sorted into chronological order. How does Java know how to do this?

String and Date both implement the Comparable interface. The Comparable interface provides a natural ordering for a class, which allows objects of that class to be sorted automatically.

`int compareTo(Object o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
public class Carta implements Comparable {
    String seme;
    String t;
    int valore;

    public Carta(String seme, int valore) {
        this.seme = seme;
        this.valore = valore;
    }

    @Override
    public int compareTo(Object o) {
        Carta c = (Carta) o;
        return this.valore - c.valore;
    }
}
```

1.10 Object ordering - comparator

```
public class Giocatore {
    List<Carta> Mano;

    public Giocatore() {
        Mano = new ArrayList();
    }

    public void sortHandByValue() {
        Comparator cmp1 = new ComparatorByValue();
        Collections.sort(Mano, cmp1);
    }

    public void sortHandBySeed() {
        Comparator cmp2 = new ComparatorBySeed();
        Collections.sort(Mano, cmp2);
    }
}
```

```
public class ComparatorBySeed implements Comparator {

    @Override
    public int compare(Object o1, Object o2) {
        Carta c1 = (Carta) o1;
        Carta c2 = (Carta) o2;
        return c1.seme.compareTo(c2.seme);
    }
}
```

```
public class ComparatorByValue implements Comparator {

    @Override
    public int compare(Object o1, Object o2) {
        Carta c1 = (Carta) o1;
        Carta c2 = (Carta) o2;
        return c1.valore - c2.valore;
    }
}
```

1.11 Listener

External listener

```
Listener lsn = new Listener(this);  
sp.addEventHandler(MouseEvent.MOUSE_CLICKED, lsn);
```

```
public class Listener implements EventHandler {  
    SimEsame w = null;  
  
    Listener(SimEsame w) {  
        this.w = w;  
    }  
  
    @Override  
    public void handle(Event t) {  
        w.addCross();  
        int id = Integer.parseInt(((Circle)t.getTarget()).getId());  
    }  
}
```

Internal listener

```
Button btn = new Button();  
btn.setText("Reset");  
Reset reset = new Reset();  
btn.addEventHandler(ActionEvent.ACTION, reset);
```

```
class Reset implements EventHandler {  
    @Override  
    public void handle(Event t) {  
        updateScore();  
    }  
}
```

Anonymous listener

```
Circle c = new Circle(r*i);  
c.setOnMouseClicked(new EventHandler<MouseEvent>() {  
    @Override public void handle(MouseEvent event) {  
        x = event.getX();  
        y = event.getY();  
    }  
});
```

Self listener

```
public class SelfLst extends Application implements EventHandler{

    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.addEventHandler(ActionEvent.ACTION, this);

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void handle(Event event) {
        System.out.println("Hi");
    }
}
```

1.12 Best practices

- **Read code**

It is impossible to become a good writer without having read books written by others.

- **Documentation**

- **Follow the defined standards**

File, functions and variable naming conventions, list of do's and don'ts, history, indentation, comments.

- **Code should be written to be reviewed**

Self review to avoid

1. Not following standards
2. Not keeping performance in mind
3. History, indentation, comments not appropriate
4. Poor readability
5. Open files not closed
6. Allocated memory not released
7. Too much hard coding
8. Too many global variables
9. Poor error handling
10. No modularity
11. Repeated code

- **Testing** is mandatory after every small or big change

- **Keep code safe** with code versioning software

- **Tools and techniques handy**

Principio di Parna il committente di una funzione deve dare all'implementatore tutte le informazioni necessarie a realizzare la funzione e nulla di più.

Chapter 2

Data structures

Classification

- **Linear:** data are in sequence
- **Non Linear**
- **Static:** the number of elements remains constant
- **Dynamic:** the number of elements can change in time
- **Homogeneous:** elements of the same type
- **Non homogenous**

2.1 Sequence

Dynamic and linear, possibly containing duplicates. To access a generic item it is necessary to scan the entire sequence either from the beginning or from the end.

2.2 Set

Collection of different elements (no duplicates are allowed). Elements do not have a position.

2.3 Dictionary

It is a key-value map.

Example: P2P systems use a DHT (distribute hash table).

2.4 Trees and graphs

Example: the base structure of a file system is a tree. Directories are the vertices while files are the leaves. Since some file systems allow to list the same

file in multiple directories by the means of symbolic links, such file systems are DAG (directed acyclic graphs).

2.5 Stack and queues

2.6 Queues with priority and disjoint sets

Chapter 3

Programming techniques

3.1 Divide et impera

3.2 Dynamic programming

3.3 Greedy

3.4 Backtrack