Mobile Application Development

Introduction to Java

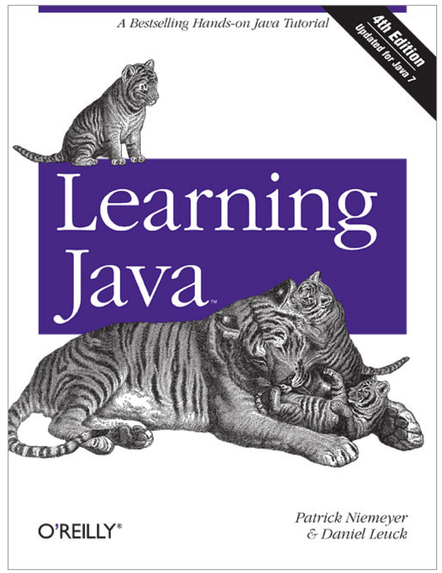
Lecture Summary

- Object-Oriented Programming
- Java basics
- Classes and objects
- Methods
- Interfaces
- OOP principles
- Practical Java
- Design patterns

References

Oracle Java Tutorials
http://docs.oracle.com/javase/tutorial/index.html

Object-Oriented Design
http://www.oodesign.com



Learning Java, 4th Edition
by Patrick Niemeyer, Daniel Leuck
O'Reilly Media - June 2013
http://shop.oreilly.com/product/0636920023463.do

Head First Design Patterns
by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra
O'Reilly Media - October 2004
http://shop.oreilly.com/product/9780596007126.do

Object-Oriented Programming

- In OOP, a program is composed of a set (graph) of interacting **objects**, representing concepts related to the specific scenario (domain)
- An object represents is a data structure which is composed by the aggregation of:
  - **state**: the actual data that the object operates on (**attributes**)
  - **behavior**: the functionalities that allow to operate with the object and its data (**methods**)
- An object can interact with another one by invoking one of its methods
- Objects have distinct responsibilities and are to be considered as independent “black-boxes”
- OOP aims at writing complex programs that are easy to manage, maintain, test, and debug
  - ... but to reach this goal, a good design is needed
  - ... and good design comes with experience!

The Java language

- Java is a general-purpose object-oriented language released by Sun Microsystems in 1995 and now owned and maintained by Oracle - currently at version 1.7
- Java programs are executed by a JVM, which decouples the *bytecode* of the program from the actual machine code, making it possible to execute the same program on different platforms (Write Once Run Everywhere)
- Java programs are based on a collection of *.class* files, which are created by compiling *.java* files (source code) with the Java compiler
- Java comes with an enormous set of libraries that support the fast and efficient development of programs, such as:
  - networking
  - working with files
  - user interface
- Java syntax is very easy to understand if you have some experience with C and C++, as they share many keywords and code-styling conventions

Java basics

- Comments

```
/* this is a
* multiline
* comment as in C/C++
*/

// this is a single-line comment
```

- javadoc Comments

```
/**
 * This comment can be used to create HTML documentation
 * with javadoc
 *
 * @author Simone Cirani
 * @version 1.0
 */
```

Java basics

- Primitive types

`boolean`	`true` or `false`
`byte`	8-bit integer
`char`	16-bit Unicode character
`double`	64-bit IEEE 754 floating point
`float`	64-bit IEEE 754 floating point
`int`	32-bit integer
`long`	64-bit integer
`short`	16-bit integer

Java basics

- Arrays

```
char[] letters = new char[26];
char[0] = 'a';
char[1] = 'b';
...
int[] primes = new int[]{2,3,5,8};
...
double matrix[][] = new double[20][10];
```

- Strings, unlike C, are objects and not arrays of chars
- Arrays are not like C arrays, they are *container objects*
- Arrays have a *length* field that can be accessed to get the size of the array

```
int size = array.length;
```

Java basics

- Conditional statements

```
if(condition){
    statement;
    statement;
    ...
}
else{
    statement;
    statement;
    ...
}
```

Java basics

- *while* loops

```
while(condition){
  statement;
  statement;
  ...
  ...
}
```

Java basics

- *do-while* loops

```
do{
  statement;
  statement;
  ...
} while(condition);
```

Java basics

- *for* loops

```
for(initialization; condition; increment){
  statement;
  statement;
  ...
}
```

Java basics

- Variable initialization and assignment

```
int i;
i = 0;
int j = i;
```

- Object creation

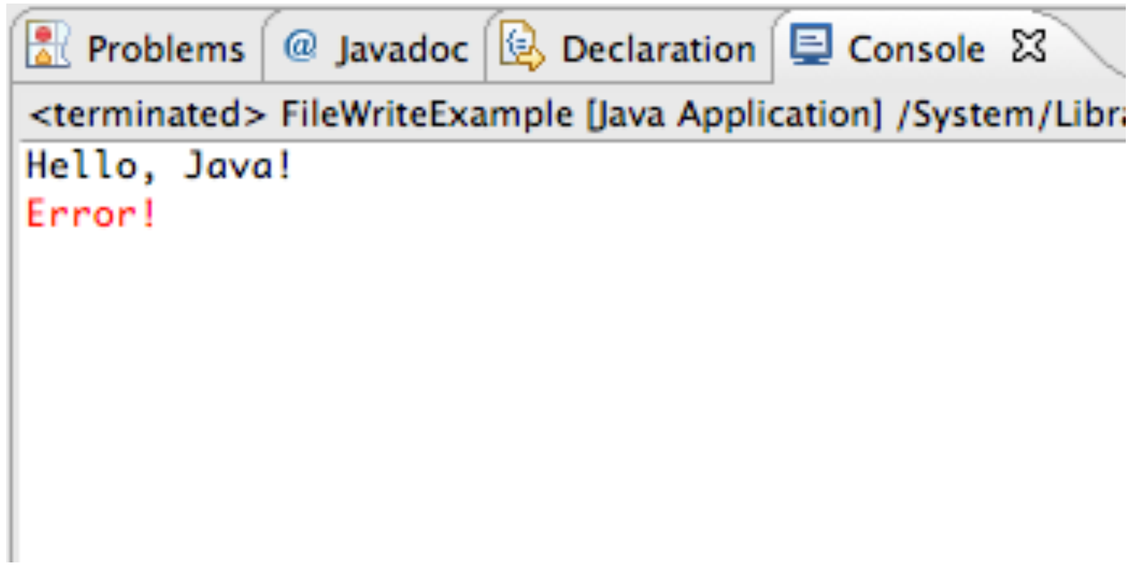
```
Object obj = new Object();
```

Java basics

- Writing to the console

```
System.out.println("Hello, Java!");
```

```
System.err.println("Error!");
```



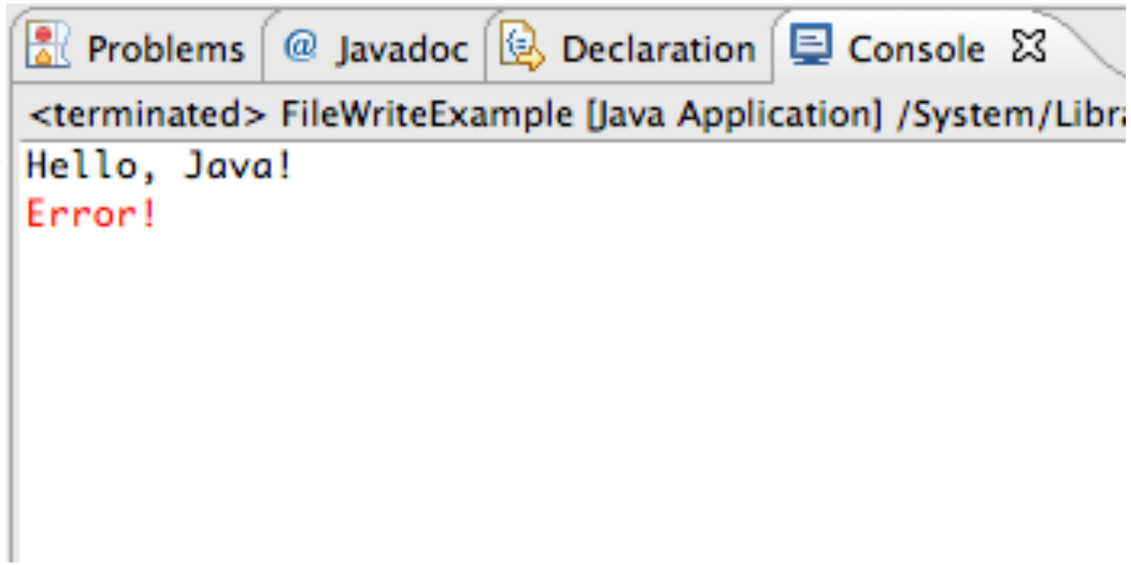
Java basics

- Writing to the console

Output stream

```
System.out.println("Hello, Java!");
```

```
System.err.println("Error!");
```

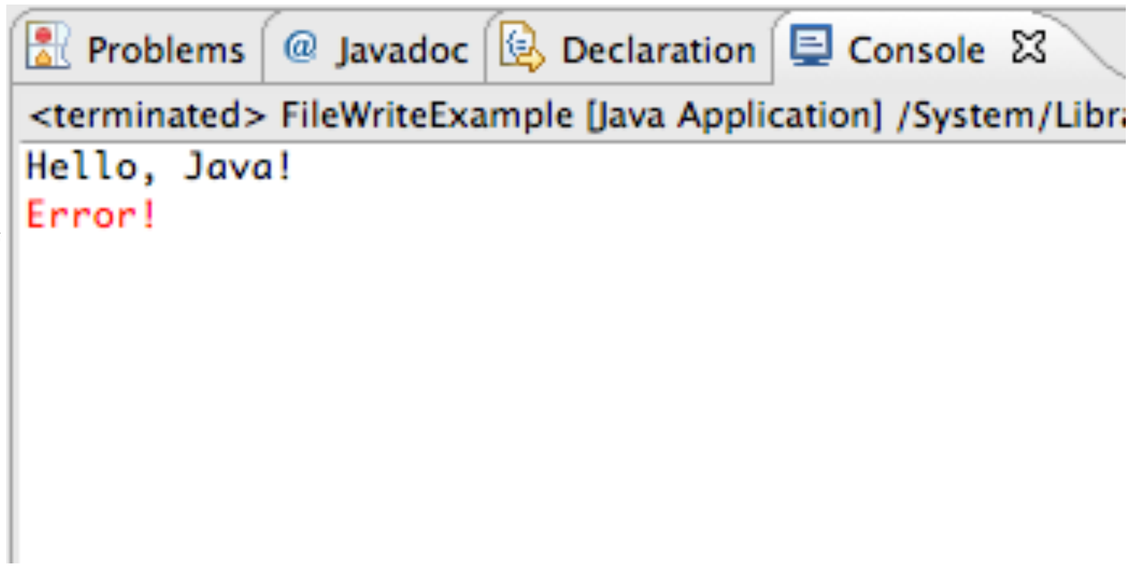


Java basics

- Writing to the console

```
System.out.println("Hello, Java!");
System.err.println("Error!");
```

Error stream



Java basics

- Running a Java program
  - a Java program starts its execution by running the `main()` method of a class
  - `main()` takes an array of String objects as arguments (program arguments)
  - in C/C++ the signature of main is `int main(int argc, char** argv)`
  - `argc = args.length`
  - `argv = args`

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

Classes and objects

- **Classes are blueprints (or prototypes) for objects**
- **Objects are instances of a class**
- A class defines the structure and behavior that all the instances of that class share
- In Java, everything is an object!
- You always have a reference to an object (like pointers in C/C++, but the * operator is implicit)
- `null` means “no reference”

Defining a class

`class` keyword defines a new class

```
class Song {

  String artist;
  String title;

  int getPlayCount(){
    ...
  }

  void resetPlayCount(){
    ...
  }

}
```

Song.java

Defining a class

```
class Song {

  String artist;
  String title;

  int getPlayCount(){
    ...
  }

  void resetPlayCount(){
    ...
  }

}
```

class name

Song.java

Defining a class

```
class Song {

  String artist;
  String title;

  int getPlayCount(){
    ...
  }

  void resetPlayCount(){
    ...
  }

}
```

attributes (or fields, or members)

Song.java

Defining a class

```
class Song {

  String artist;
  String title;

  int getPlayCount(){
    ...
  }

  void resetPlayCount(){
    ...
  }

}
```

methods

Song.java

Defining a class

return type

```
class Song {

  String artist;
  String title;

  int getPlayCount(){
    ...
  }

  void resetPlayCount(){
    ...
  }

}
```

Song.java

Accessing instance variables

- It is possible to access the instance variables of an object using the dot operator (similar to C structs)

```
String a = song.artist;
String t = song.title;
```

- Accessing an instance variable of a `null` object raises a NullPointerException at runtime!

Invoking methods

- The dot operator is used also to invoke a method of an object

```
song.getPlayCount();
song.resetPlayCount();
```

- Invoking a method of a `null` object raises a NullPointerException at runtime!
- For methods that take arguments:
  - primitive types are passed by value
  - objects are passed by reference

Static members

- The keyword `static` before a member means that the member is not related to the specific instance but to the class itself
- Static members are shared among all instances of the class
- Class constants should typically be static members
- Static members (either fields or methods) can be invoked using the dot notation against the class name

```
public class Song {
   String artist;
   String title;
   static int MAX_LENGTH = 3600;

   int getPlayCount(){

   }

   void resetPlayCount(){

   }
}
```

```
int i = Song.MAX_LENGTH;
```

Local variable scoping

- As in C and C++, in Java, scope is determined by the placement of curly braces

Visibility of `i`

```
int i = 10;
{
    int j = 0;
    {
      i = 1;
      int k = 100;
    }
}
{
    int j = 2;
}
i = 0;
```

Visibility of `k`

Visibility of `j`

Visibility of `j`

this

- `this` is a reference to the current object: it basically means “myself”
- The `this` keyword is used to explicitly reference an instance variable, rather than a local variable

```
void f(){
    String artist = "Muse";
    this.artist = artist;
}
```

Instance variable artist

Local variable artist

Creating objects

- Objects are created using a constructor
- A constructor is a special method that takes the same name of the class and has no return type
- Constructors are used to initialize instance variables so that, when the object is used, it is in a consistent state
- A constructor with no arguments is called **default constructor**:

```
Song(){
    this.artist = null;
    this.title = null;
}
```

- Constructors can take arguments:

```
Song(String artist, String title){
    this.artist = artist;
    this.title = title;
}
```

Constructor overloading

- More than one constructor can be defined for a class
- It is possible for a constructor to invoke another constructor using the `this()` constructor
- The constructor being called is the one with the same number of arguments

```
Song(String artist, String title){
    this.artist = artist;
    this.title = title;
}

Song(String artist){
    this(artist,null);
}

Song(){
    this(null,null);
}
```

Creating objects

- Constructors are called with the new operator:

```
Song song = new Song("Muse","Resistance");
```

Destroying objects

- In Java, objects are not explicitly destructed
- Memory management is automatic
- A technique called **garbage collection** is used to automatically destroy objects that are no longer needed, as they are not referenced anywhere else in the code
- In Java, there is no `delete` keyword

Inheritance

- **Inheritance** is a fundamental principle of Object-Oriented Programming
- Inheritance allows to define new functionalities of a class by **extending** it
- The class that is being extended is called superclass
- The process of extending a class is called **subclassing**
- A class inherits all the variables and methods defined in its superclass
- All classes therefore define a hierarchy which ultimately defines which functionalities the instances of each class have
- Everything is an **Object**! One class, called Object, is the root of the hierarchy
- Inheritance creates a semantic “is-a” relationship
- Anything a superclass can do, its subclasses can do

Inheritance

- Suppose we want to extend the functionalities of our `Song` class for a new `MP3Song` class, which has also a `bitrate` information and a `resample()` method
- All we have to do is to create a new class called `MP3Song` and declare the subclassing of the `Song` class with the `extends` keyword

```
class MP3Song extends Song {

    int bitrate;

    void resample(){

    }

}
```

the `MP3Song` class subclasses `Song` the class

MP3Song.java

Inheritance

- Since a `MP3Song` **is-a** `Song`, it is perfectly fine to invoke `getPlayCount()` and `resetPlayCount()`
- Additionally, we can also invoke the `resample()` method on a `MP3Song` instance
- This way, we have extended the behavior of our base (super) class
- The `super` keyword is used to access a field or invoke a method explicitly on the superclass

```
class MP3Song extends Song {

   int bitrate;

   void resample(){

   }

}
```

MP3Song.java

Subclass constructors

- The constructor of a subclass should always invoke the constructor of its superclass, so that the initialization of the basic variables is consistent
- In order to invoke the superclass constructor, we use the special constructor `super()`
- `super()` follows the same rules of `this()` for the number of arguments

```
MP3Song(String artist, String title, int bitrate){
    super(artist,title);
    this.bitrate = bitrate;
}
```

Encapsulation

- **Encapsulation** is the second fundamental principle of Object-Oriented Programming
- Encapsulation refers to hiding your class data from the outside and to expose only the data that should be really accessible from the outside
- Why? Let's take a look at our `Song` class...

```
class Song {

   String artist;
   String title;

   ...

   Song(String artist, String title){
      this.artist = artist;
      this.title = title;
   }

}
```

Song.java

```
...
Song song = new Song("Rammstein","Ich Will");
...
String artist = song.artist;
...
```

<someone else's class>.java

Encapsulation

- Suppose we need to change our **Song** class and now the field `artist` should be named `a`
- The code using **Song** will break!
- Another problem is that since the fields are directly accessible, they are writable even though we could wish to prevent such behaviors (read-only fields)
- This happens because the **Song** class was exposing too many details about its implementation on the outside and other parts of the code were accessing the data directly

```
class Song {

   String a;
   String t;


   ...

   Song(String artist, String title){
      this.a = artist;
      this.t = title;
   }

}
```

Song.java

```
...
Song song = new Song("Rammstein","Ich Will");
...
String artist = song.artist;
...
```

<someone else's class>.java

Encapsulation

- The right way to avoid this is through encapsulation
  - fields should be hidden from outside the class
  - access to the fields should occur through proper methods called getter/setter
  - but how can we hide the fields of our class?
- Java defines **access modifiers**, special keywords that are used to set the visibility of fields and methods in the code:
  - `private`: accessible only within the class itself
  - `protected`: accessible only within the class and its subclasses
  - `public`: accessible from anywhere
  - default (none): accessible only within the class and its subclasses and by the classes in the same **package**

Encapsulation

- The solution is to set the fields as `private` and then implement a getter and a setter for each
- Access to the fields is performed through these methods
- Changes to the class fields does not affect the methods (and how they are invoked, so other code is OK)
- Classes should be used as black-boxes: there is no need to have the details of the implementation as long as we know that we can do something

Encapsulation

```
class Song {

   private String artist;
   private String title;

   Song(String artist, String title){
      this.artist = artist;
      this.title = title;
   }
   ...
   public String getArtist(){
      return this.artist;
   }

   public void setArtist(String artist){
      this.artist = artist;
   }

   public String getTitle(){
      return this.title;
   }
   ...
}
```

Song.java

```
...
Song song = new Song("Rammstein","Ich Will");
...
String artist = song.getArtist();
...
```

<someone else's class>.java

if we change the name of the fields, we just need to change the `getArtist()` method, but the code on the right still works fine

the `title` field is read-only since it only exposes a getter

POJOs

- POJO stands for Plain Old Java Object
- POJOs are instances of classes that provide
  - a default constructor
  - a getter and a setter for each field
- Conventions (pay attention to uppercase!):
  - getter for attribute of type T and name attribute: `public T getAttribute()`
  - setter for attribute of type T and name attribute: `public void getAttribute(T attribute)`

```
public class Pojo {

   private int a;
   private int b;

   public Pojo(){}

   public int getA() {
      return a;
   }

   public void setA(int a) {
      this.a = a;
   }

   public int getB() {
      return b;
   }

   public void setB(int b) {
      this.b = b;
   }

}
```

Interfaces

- Java **interfaces** provide a way to specify the behavior of objects without actually implementing this behavior
- Behavior is specified by a set of methods
- Classes that provide a concrete behavior for an interface are said to **implement the interface**
- For instance, we can define a `Playable` interface which defines two methods `play()` and `stop()`
- The `Playable` interface can be implemented by a `CDPlayer` class or an `MP3Player` class
- Interfaces can be used by classes to declare that they are capable to perform some behavior
- Interfaces can be extended by other sub-interfaces, exactly as with classes

Interfaces

- Interfaces are declared as classes, but require the usage of the `interface` keyword
- Interfaces do not have instance variables since they are not classes
- Interfaces only specify behavior by defining methods
- Interface methods do not have a concrete implementation
- The implementation of interface methods is left to concrete classes
- Interfaces are useful when the code does not really depend on how a class does something, but rather it is important to rely on the fact that the class can do something

```
public interface Playable {

   public void play();
   public void stop();

}
```

Playable.java

Interfaces

```
public class MusicPlayer implements Playable {

  ...

  public void play() {
    System.out.println("playing...");
  }

  public void stop() {
    System.out.println("stopped!");
  }

}
```

MusicPlayer.java

Polymorphism

- **Polymorphism** is the third fundamental principle of Object-Oriented Programming
- With inheritance, all the subclasses inherit the same methods of their superclass
- It is possible for subclasses to override the methods of their superclasses
- Since we can always treat an object as a member of one of its superclasses, the actual behavior that it will perform when one of its methods is invoked depends on its concrete type
- The ability to adapt the behavior to the concrete type is called polymorphism

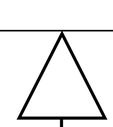
Polymorphism

```
public class Animal {

   public Animal(){}

   public void eat(){
      System.out.println("What do I eat?");
   }

}
```



```
public class Cat extends Animal {

   public Cat(){}

   public void eat(){
      System.out.println("I eat what a cat eats");
   }

}
```

```
public class Dog extends Animal {

   public Dog(){}

   public void eat(){
      System.out.println("I eat what a dog eats");
   }

}
```

Polymorphism

```
Animal a = new Animal();
a.eat();
Animal c = new Cat();
c.eat();
Animal d = new Dog();
d.eat();
```

```
What do I eat?
I eat what a cat eats
I eat what a dog eats
```

Polymorphism

- It is possible to treat an object as one of its superclasses
- It is also possible to treat an object as one of its implemented interfaces

Callbacks through interfaces

- Interfaces are often used to implement **callback methods**
- A callback method is a method that is passed as an argument of another method to be later invoked
- In C/C++ this can be made with function pointers
- Callback methods are useful to handle events that can occur at an unspecified time (asynchronous events)
- For instance, user interface components that receive input events from the user can specify a callback method to be executed when some UI-related event occurs

Callbacks through interfaces

```
public interface ClickListener {

    public void onButtonClicked(Button button);

}
```

The `ClickListener` interface defines a method that will be triggered when a click event will be received by a button

Callbacks through interfaces

```
public class ViewController implements ClickListener {

   public ViewController(){
      Button button = new Button("Click me!");
      button.setClickListener(this);
   }

   private void showAlert(String text){
      ...
   }

   public void onButtonClicked(Button button) {
      this.showAlert(button.getText());
   }

}
```

this class implements the `ClickListener` interface and sets itself as listener for click events related to the button

Callbacks through interfaces

```
public class ViewController implements ClickListener {

   public ViewController(){
      Button button = new Button("Click me!");
      button.setClickListener(this);
   }

   private void showAlert(String text){
      ...
   }

   public void onButtonClicked(Button button) {
      this.showAlert(button.getText());
   }

}
```

this is the callback method that will be triggered

Callbacks through interfaces

```
public class Button {

   private String text;
   private ClickListener clickListener;

   public Button(String text){
      this.text = text;
   }

   public void setClickListener(ClickListener clickListener){
      this.clickListener = clickListener;

   }

   public String getText(){
      return this.text;
   }

   ...
   this.clickListener.onButtonClicked(this);
   ...

}
```

a click event has occurred, so the listener should be notified

Abstract classes

- Abstract classes are classes that cannot be instantiated, even though they can have constructors
- They define basic state and behavior but need subclasses to implement abstract methods
- Abstract methods are marked with the `abstract` keyword and have no implementation, as for interface methods

```
public abstract class MyClass {

  public MyClass(){}

  public void f(){
    ...
  }

  public abstract void g();

}
```

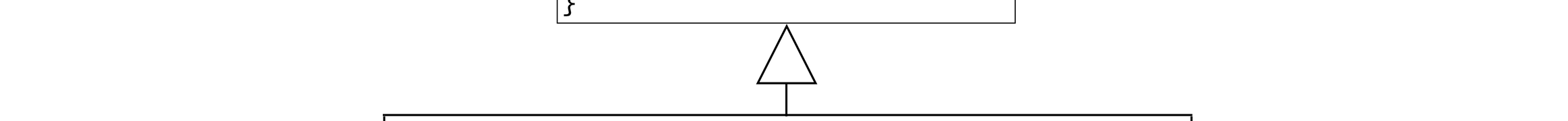
Abstract classes

```
public abstract class Animal {

   public Animal(){}

   public abstract void eat();

}
```



```
public class Cat extends Animal {

   public Cat(){}

   public void eat(){
      System.out.println("I eat what a cat eats");
   }

}
```

```
public class Dog extends Animal {

   public Dog(){}

   public void eat(){
      System.out.println("I eat what a dog eats");
   }

}
```

Abstract classes

```
Animal a = new Animal();
```

Abstract classes

```
Animal a = new Animal();
```

FORBIDDEN!!! Abstract classes cannot be instantiated!

Abstract classes

```
Animal a = new Animal();
```

FORBIDDEN!!! Abstract classes cannot be instantiated!

```
Animal c = new Cat();
c.eat();
Animal d = new Dog();
d.eat();
```

```
I eat what a cat eats
I eat what a dog eats
```

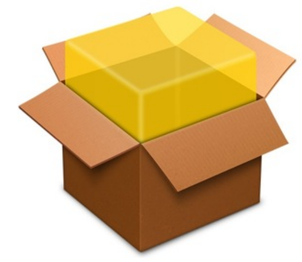
Good practices

- A good principle for better OOP code is to define classes and interfaces in such a way that:
  - classes are very specialized in doing a few things
  - classes expose only the methods that must be used in order to interact with them
  - classes can be treated as “black-boxes” that provide some functionality
- Avoid classes that do too many things: they can be probably split into more classes!
- Always try to define interfaces that are useful for your problem: it is much easier to think about what objects can do, rather than having to think about the functionalities that they inherit

`final`

- `final` can be used:
  - to set a variable as constant
  - to avoid the override of a method
  - to avoid the extension of a class (it is the end of the hierarchy)

Packages



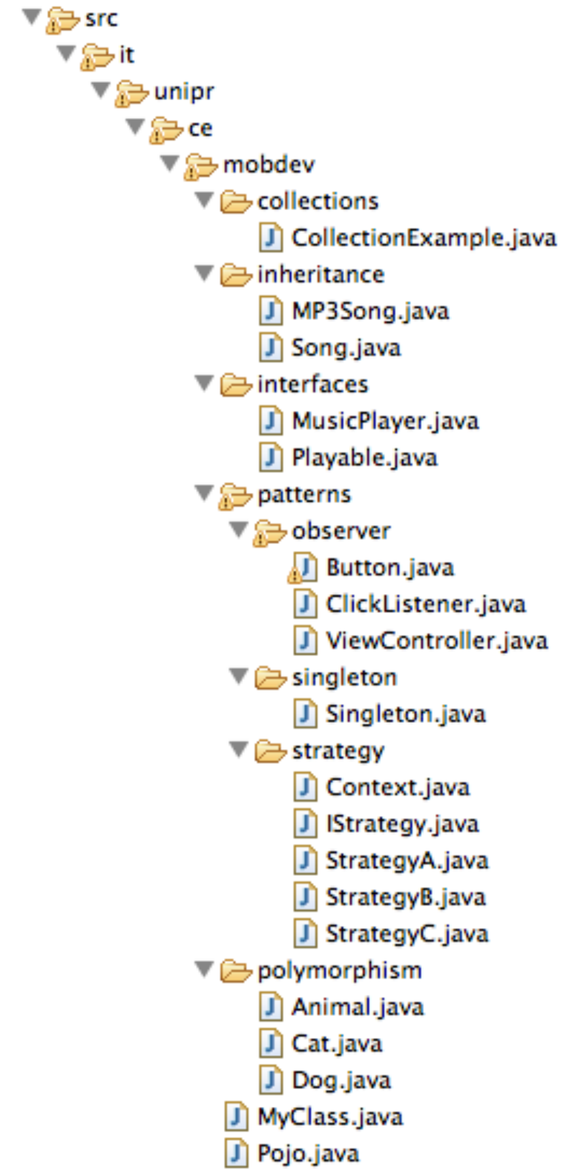
- Classes and interfaces are organized in **packages**
- Packages are similar to folders where you keep files, indeed packages translate to a folder structure in the filesystem
- Packages organize Java classes into namespaces
- Classes and interfaces in the same package usually share similar functionalities and goals
- Java already provides a huge number of well-documented[1,2] packages with different objectives, such as networking, working with files, ...
- Packages make it possible to define more classes with the same name, as long as they belong to a different package (disambiguation of names)

[1] http://docs.oracle.com/javase/7/docs/api/

[2] http://docs.oracle.com/javase/6/docs/api/

Packages

- Packages are usually named by reversing your domain: `it.unipr.ce.mobdev`
- The package name translates into a folder structure
- Java source and class files are stored in the folder corresponding to the package path



Packages

- Each Java source file declares to which package it belongs with the `package` keyword, at the beginning of the file

```
package it.unipr.ce.mobdev;

public class MyClass {

    ...

}
```

`import`

- The `import` keyword is used to include a single class or interface or an entire package into the current class or interface
- Importing is needed in order to use the classes and interfaces defined somewhere else

```
import java.util.List;
```
⟶ Import class `List` of package `java.util`

```
import java.io.*;
```
⟶ Import all classes of package `java.io`

import

- Importing is not needed if you use:
  - classes and interfaces defined in the java.lang package (always accessible)
  - classes and interfaces defined in the same package
  - classes and interfaces by specifying the fully qualified name of the class

```
it.unipr.ce.mobdev.MyClass c = new it.unipr.ce.mobdev.MyClass();
```

Collections

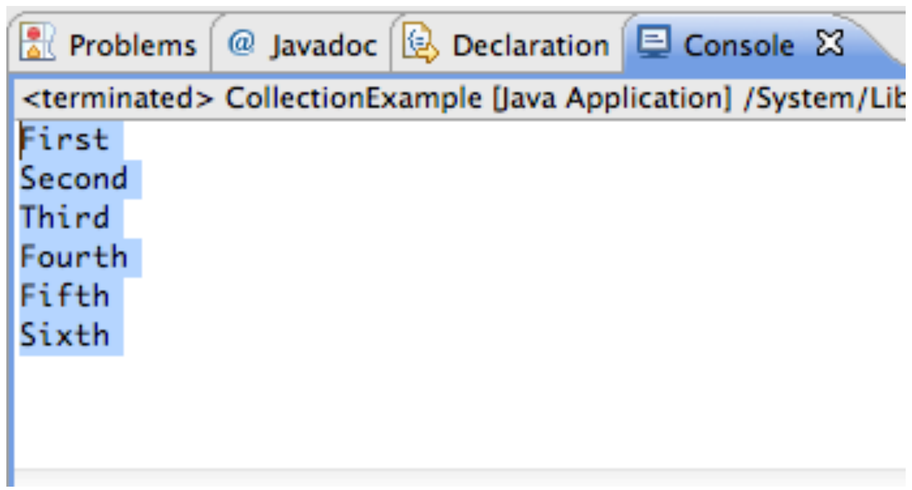
- A collection, sometimes called a container, is simply an object that groups multiple elements into a single unit
- Collections are used to store, retrieve, manipulate, and communicate aggregate data
- Java provides many classes to work with collections, defined in the `java.util` package:
  - `List<T>`: `ArrayList<T>`, `Vector<T>` are used as ordered list of objects
    - main methods: `add(T)`, `get(int)`, `size()`
  - `Map<K,V>`: `HashMap<K,V>`, `TreeMap<K,V>` are used as key/value stores
    - main methods: `put(K,V)`, `get(K)`, `size()`

Iterating over collections

- In order to iterate through a collection, an Iterator object is used (`java.util.Iterator`)
- An iterator is an object that allows to step through collections
- Iterators provide methods to check if the collection has more elements, to retrieve the next element, to remove the current element:
  - `public boolean hasNext()`: returns true if other elements are in the collection
  - `public T next()`: retrieves the next element in the collection
  - `public void remove()`: remove the last object returned by `next()`
- Iterators are the most correct and safe way to traverse collections

Collections example

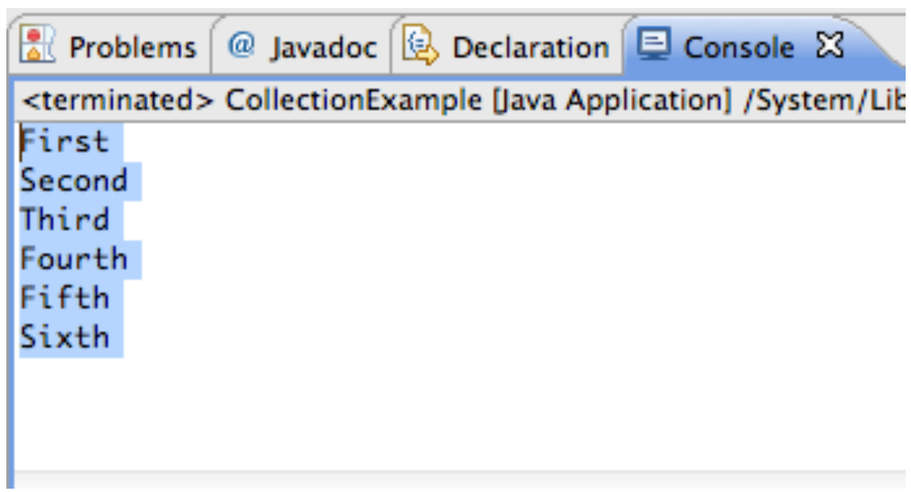
```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```



Collections example

```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```

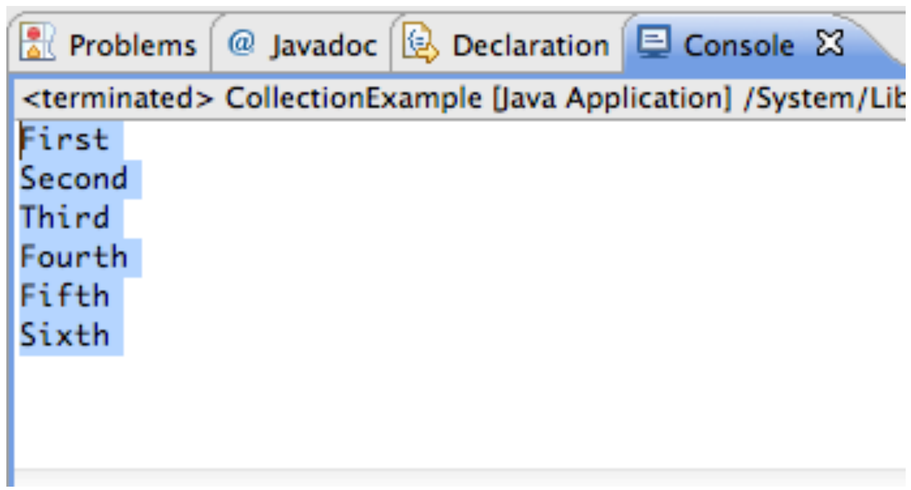
a list of String objects



Collections example

```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```

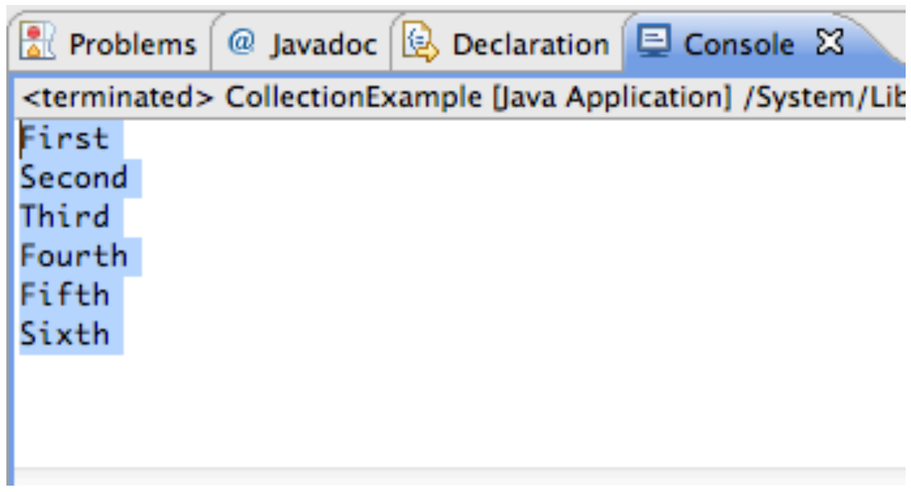
the `add()` method is used to add an element to the list



Collections example

```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```

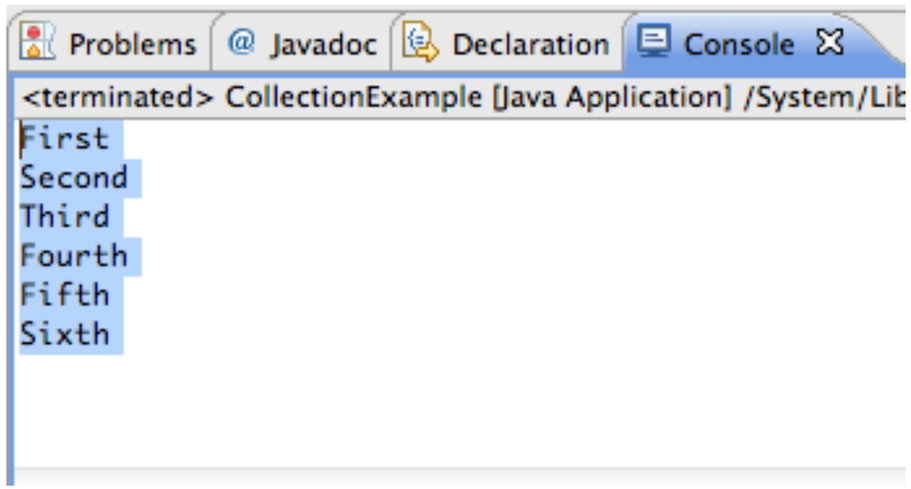
an Iterator for String objects



Collections example

```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```

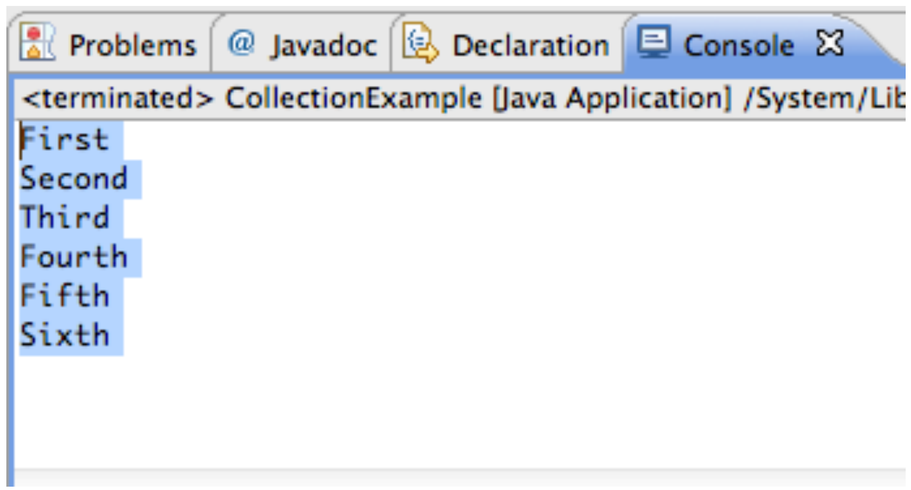
we get an Iterator from the list



Collections example

```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```

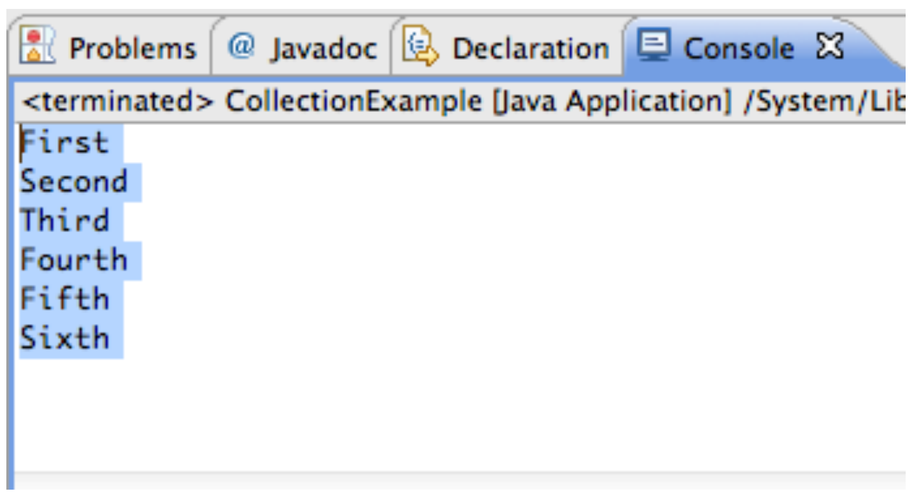
we loop as long as there are elements in the list



Collections example

```
java.util.ArrayList<String> list = new java.util.ArrayList<String>();
list.add("First");
list.add("Second");
list.add("Third");
list.add("Fourth");
list.add("Fifth");
list.add("Sixth");
java.util.Iterator<String> it = list.iterator();
while(it.hasNext()){
   String element = it.next();
   System.out.println(element);
}
```

we get the element with the `next()` method

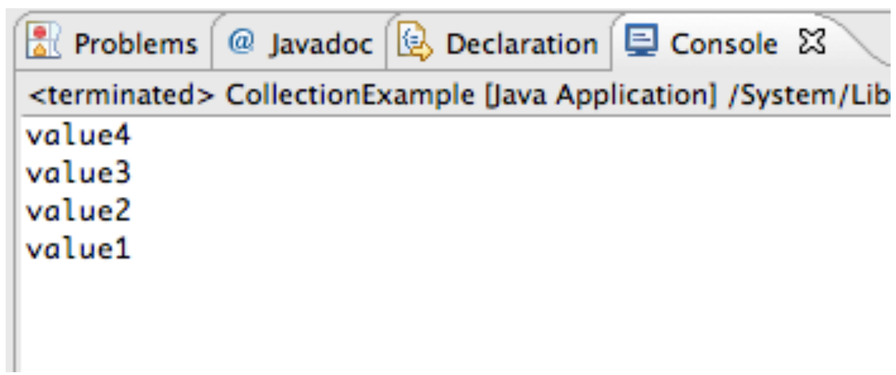


Fast enumeration

```
java.util.HashMap<String,String> map = new java.util.HashMap<String,String>();


map.put("key1","value1");
map.put("key2","value2");
map.put("key3","value3");
map.put("key4","value4");


for(String value : map.values()){
   System.out.println(value);
}
```



Fast enumeration

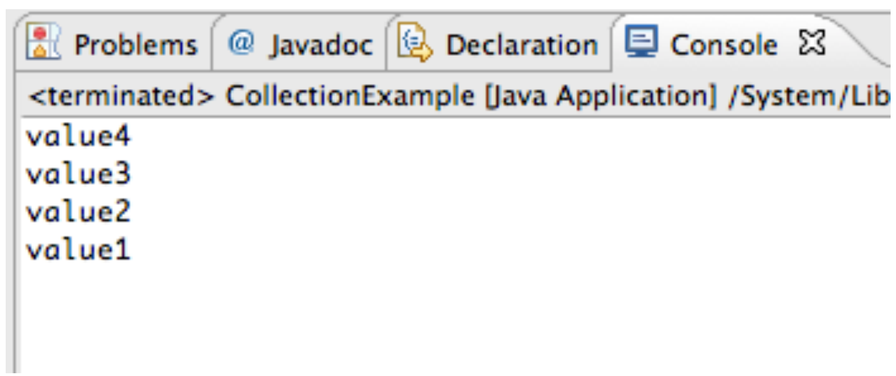
```
java.util.HashMap<String,String> map = new java.util.HashMap<String,String>();


map.put("key1","value1");
map.put("key2","value2");
map.put("key3","value3");
map.put("key4","value4");


for(String value : map.values()){
   System.out.println(value);
}
```

a map that stores string/string pairs

when using a HashMap there is no guarantee on order (no relation between insertions and extractions)



Fast enumeration

```
java.util.HashMap<String,String> map = new java.util.HashMap<String,String>();


map.put("key1","value1");
map.put("key2","value2");
map.put("key3","value3");
map.put("key4","value4");


for(String value : map.values()){

   System.out.println(value);

}
```

we can iterate over a collection
without an explicit iterator

Working with files

- The `java.io` and `java.nio` packages include classes that allow to work with files and streams (flows of data), both of bytes and characters, easily
- The package includes the following prominent classes:

`File`	Encapsulates access to a file or directory in the file system
`InputStream`	Abstract class that defines the basic functionality for reading bytes from a stream
`OutputStream`	Abstract class that defines the basic functionality for writing bytes to a stream
`FileInputStream`	Class for reading bytes from a file
`FileOutputStream`	Class for writing bytes to a file
`Reader`	Abstract class for reading characters from a stream
`Writer`	Abstract class for writing characters to a stream

Reading from a file

```
File file = new File("logs/README.txt");
if(file.exists()){
  try {
    BufferedReader input =  new BufferedReader(new FileReader(file));
    String line = null;
    while (( line = input.readLine()) != null){
      System.out.println(line);
    }
    input.close();
  } catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }
}
else{
  System.err.println("File does not exist!");
}
```

Writing to a file

```
File file = new File("logs/WRITEME.txt");
try {
  if(file.exists() == false) file.createNewFile();
  BufferedWriter output =  new BufferedWriter(new FileWriter(file));
  String line = "I am writing this to a file";
  output.write(line);
  output.close();
} catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}
```

Networking

- Java support for networking resides in the `java.net` package
- The package includes the following prominent classes:

`DatagramSocket`	UDP socket for sending and receiving packets
`DatagramPacket`	UDP packet to be sent or received
`ServerSocket`	Server TCP socket
`Socket`	Client TCP Socket
`URLConnection`	Communication link between the application and a URL (abstract class)
`HttpURLConnection`	`URLConnection` with support of HTTP
`URL`	Uniform Resource Locator, a pointer to a resource

HttpURLConnection example

```
try {
   URL url = new URL("http://mobdev.ce.unipr.it/2013");
   HttpURLConnection hc = (HttpURLConnection)url.openConnection();
   BufferedReader in = new BufferedReader(new InputStreamReader(hc.getInputStream()));
   String inputLine;
   while ((inputLine = in.readLine()) != null){
      System.out.println(inputLine);
   }
   in.close();
} catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
}
```

Threads & Concurrency

- A thread is a flow of control within a process
- Multiple threads can be executed concurrently within a program to perform several task “in parallel”
- Parallelization is only conceptual as the execution of each thread is actually performed by the CPU(s)
- Support for multi-threading in Java is native in the language
- The `java.lang` package defines the **`Thread`** class and the **`Runnable`** interface

`Thread`	A thread of execution within a Java program
`Runnable`	Interface that must be implemented by any class whose instances are to be executed within a Thread

Threads & Concurrency

- Typically, when some code needs to be executed within a thread, that code is going to be wrapped in a class that implements the `Runnable` interface
- The `Runnable` interface requires to implement the method `run()`
- The thread dies when the `run()` method returns
- Once the class that implements the `Runnable` interface has been implemented, it can be used to create `Thread` instances by passing it as the argument of the constructor
- The `Thread` starts its execution by invoking the `start()` method, which in turns invokes the `run()` method of the `Runnable`
- Sometimes, `Runnable` interface is implemented inline, using *anonymous classes*

Threads Example

```
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name){
      this.name = name;
   }

   public String getName(){
      return this.name;
   }

   public void run() {
      for(int i = 0; i < 10; i++){
         System.out.println(this.name);
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

}
```

Threads Example

```
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name){
      this.name = name;
   }

   public String getName(){
      return this.name;
   }

   public void run() {
      for(int i = 0; i < 10; i++){
         System.out.println(this.name);
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

}
```

The **`Runnable`** interface must be implemented

Threads Example

```
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name){
      this.name = name;
   }

   public String getName(){
      return this.name;
   }

   public void run() {
      for(int i = 0; i < 10; i++){
         System.out.println(this.name);
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

}
```

```
MyRunnable r1 = new MyRunnable("Thread-1");
MyRunnable r2 = new MyRunnable("Thread-2");
Thread t1 = new Thread(r1);
Thread t2= new Thread(r2);
t1.start();
t2.start();
```

Thread instances are created by passing a **Runnable** as an argument

Threads Example

```
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name){
      this.name = name;
   }

   public String getName(){
      return this.name;
   }

   public void run() {
      for(int i = 0; i < 10; i++){
         System.out.println(this.name);
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

}
```

```
MyRunnable r1 = new MyRunnable("Thread-1");
MyRunnable r2 = new MyRunnable("Thread-2");
Thread t1 = new Thread(r1);
Thread t2= new Thread(r2);
t1.start();
t2.start();
```

1

`Thread` begins its execution by invoking the `start()` method

Threads Example

```
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name){
      this.name = name;
   }

   public String getName(){
      return this.name;
   }

   public void run() {
      for(int i = 0; i < 10; i++){
         System.out.println(this.name);
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

}
```

```
MyRunnable r1 = new MyRunnable("Thread-1");
MyRunnable r2 = new MyRunnable("Thread-2");
Thread t1 = new Thread(r1);
Thread t2= new Thread(r2);
t1.start();
t2.start();
```

2

Thread invokes the **run()** method of the **Runnable**

Threads Example

```
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name){
      this.name = name;
   }

   public String getName(){
      return this.name;
   }

   public void run() {
      for(int i = 0; i < 10; i++){
         System.out.println(this.name);
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

}
```

```
MyRunnable r1 = new MyRunnable("Thread-1");
MyRunnable r2 = new MyRunnable("Thread-2");
Thread t1 = new Thread(r1);
Thread t2= new Thread(r2);
t1.start();
t2.start();
```

Markers | Properties | Servers | Data Source Explorer | Snippets | Console

<terminated> MyRunnable [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/C

```
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
Thread-2
Thread-1
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
```

there is no guarantee on how threads are scheduled and executed

Threads & User Interfaces

- It is very important to use threads: if there are tasks that need to be performed in background so that the application remains responsive
- User interface tasks MUST be executed in the main thread so that user input events are always captured promptly
- It is forbidden to execute user interface updates on secondary threads
- All long-running tasks, such as network activities (it does not matter how much data are going to be transferred, there is no guarantee on the bit rate!), reading from big files, etc..., MUST be executed in secondary threads

Design Patterns

- Design Patterns are documented best practices that can be used as models when designing software
- Design Patterns define how specific problems can be addressed as reusable solutions
- Object-Oriented Programming is about relationships and interactions among objects that belong to a domain
- Good OOP techniques can help write **better**, **reusable**, and **maintainable** code **faster**
- Design Patterns are very effective tools as they can be used to say a lot about the code without worrying to give details
- Design Patterns can be applied basically to all OOP languages
- “Do not waste time reinventing the wheel, someone has already done it and it works just fine!”

Design Patterns

- There are several design patterns that are used extensively in frameworks such as Android and iOS
- The hard part about design patterns is to recognize when to use and which one does really help solving our problem
- Do not ALWAYS use them, use them only when there is a real advantage
- You have already seen one design pattern today when we talked about callback methods: **observer** pattern
- We will go through a few examples of very useful design patterns and their Java implementation

Singleton

- Problem:
  - [P1] Ensure that only one instance of a class is created
  - [P2] Provide a global point of access to the object

?

Singleton

- Problem:
  - [P1] Ensure that only one instance of a class is created
  - [P2] Provide a global point of access to the object

Solution

```
public class Singleton {

   private static Singleton instance = null;

   private Singleton(){}

   public static Singleton getInstance(){
      if(instance == null) instance = new Singleton();
      return instance;
   }

}
```

Usage

```
Singleton s = Singleton.getInstance();
```

Singleton explained

```
public class Singleton {

   private static Singleton instance = null;

   private Singleton(){}

   public static Singleton getInstance(){
      if(instance == null) instance = new Singleton();
      return instance;
   }

}
```

The `getInstance()` is a `public` and `static` method so it is visible and accessible from everywhere just by using the class name [P2]

Singleton explained

```
public class Singleton {

   private static Singleton instance = null;

   private Singleton(){}

   public static Singleton getInstance(){
      if(instance == null) instance = new Singleton();
      return instance;
   }

}
```

The `instance` field is `static` so it is shared among all the instances of the class; also, it is also `private` so it is accessible only within the class

Singleton explained

```
public class Singleton {

   private static Singleton instance = null;

   private Singleton(){}

   public static Singleton getInstance(){
      if(instance == null) instance = new Singleton();
      return instance;
   }

}
```

The constructor is `private` so it is accessible only within the class

Singleton explained

```
public class Singleton {

   private static Singleton instance = null;

   private Singleton(){}

   public static Singleton getInstance(){
      if(instance == null) instance = new Singleton();
      return instance;
   }

}
```

The constructor is called only if the `static instance` field is `null`, so only the first time that the `getInstance()` method is invoked. All other times the same instance object is returned [P1]

Strategy

- Problem:
  - Define a family of algorithms, encapsulate each one, and make them interchangeable

Strategy

```
public interface IStrategy {

   public void doStrategy();

}
```

```
public class StrategyA implements IStrategy {

   public StrategyA(){}

   public void doStrategy() {
      /* strategy A */
   }

}
```

```
public class StrategyB implements IStrategy {

   public StrategyB(){}

   public void doStrategy() {
      /* strategy B */
   }

}
```

Strategy

The `IStrategy` interface defines a method `doStrategy()` which must be implemented

```
public interface IStrategy {

   public void doStrategy();

}
```

```
public class StrategyA implements IStrategy {

   public StrategyA(){}

   public void doStrategy() {
      /* strategy A */
   }

}
```

```
public class StrategyB implements IStrategy {

   public StrategyB(){}

   public void doStrategy() {
      /* strategy B */
   }

}
```

Strategy

```
public interface IStrategy {

   public void doStrategy();

}
```

The `StrategyA` and `StrategyB` classes implement the `IStrategy` interface their way

```
public class StrategyA implements IStrategy {

   public StrategyA(){}

   public void doStrategy() {
      /* strategy A */
   }

}
```

```
public class StrategyB implements IStrategy {

   public StrategyB(){}

   public void doStrategy() {
      /* strategy B */
   }

}
```

Strategy

```
public class Context {

  private IStrategy strategy;

  public Context(){}

  public void setStrategy(IStrategy strategy){
    this.strategy = strategy;
  }

  public void method(){
    /*...*/
    strategy.doStrategy();
    /*...*/
  }

}
```

Strategy

```
public class Context {

   private IStrategy strategy;

   public Context(){}

   public void setStrategy(IStrategy strategy){
      this.strategy = strategy;
   }

   public void method(){
      /*...*/
      strategy.doStrategy();
      /*...*/
   }

}
```

The `Context` class needs a class that implements the `IStrategy` interface to execute the `method()` method, but it does not care about the actual implementation

Strategy

The `setStrategy()` method is used to change at runtime the actual strategy

```
public class Context {

   private IStrategy strategy;

   public Context(){}

   public void setStrategy(IStrategy strategy){
      this.strategy = strategy;
   }

   public void method(){
      /*...*/
      strategy.doStrategy();
      /*...*/
   }

}
```

Strategy

```
Context context = new Context();
context.setStrategy(new StrategyA());
context.method();
```

Strategy

```
Context context = new Context();
context.setStrategy(new StrategyA());
context.method();
```

Now the `context` object uses `StrategyA`'s implementation of the `doStrategy()` method

Strategy

- The Strategy pattern allows to change parts of an algorithm at runtime, as long as they implement the interface required for the execution of that portion of code
- Using the Strategy pattern, we can ignore the existence of actual implementations
- Example:
  - at some point in our code, we want to store some data but we do not care whether the data are going to be stored in a database, in a file, or in the cloud (these might be user's preferences)
  - we just need to define an `IStorer` interface with a `store()` method and let implementors decide how to store data (the strategy)
  - the rest of our code will remain the same, we are just relying on the fact that someone will set a proper `IStorer` implementation to do the storage

Good practices

- As a general rule, you should always prefer using interfaces when designing classes where some behavior might change over time
- In other words, always **favor composition over inheritance!**
- Composition means that you let the user of your class set the behavior at runtime by providing a concrete implementation of the interfaces that you use
- Inheritance creates a tight coupling, which makes classes very dependent on each other, and therefore hard to maintain
- Any time you type `new` you are making a hard decision that is going to last forever!
- Before typing `new`, always think if this is the right thing to do (always answer to this question: "Am I supposed to take this decision or should I let others decide?")
- Always **minimize the interdependence among your classes!**

Mobile Application Development

Introduction to Java