University of Liège

Object-oriented programming

INFO0062-1

OOP Synthesis

Julien GUSTIN

February 1, 2021

Contents

1 Chapter 1: OOP Approach

1.1 Object

In oop, **Object** are data structure that are present in memory during program execution. It combines data and executable code. This code corresponds to the object than can be performed by the object.
An object has :

1. An interface Characterizes each method by a header, composed of
   - a method name,
   - a list of parameters >= 0
   - the return type
2. An implementation Of an object contains
   - its variables
   - the body of its methods

The execution of a program is seen as a sequence of interactions between obects (call messages)

1.2 Classes and Objects

A **class** is a sets of features of the object. A class can be **instantiated** in order to construct an object, an object is an instance of a class. (create a new object and return his reference)
Example :

```
ClassName obj = new ClassName();
```

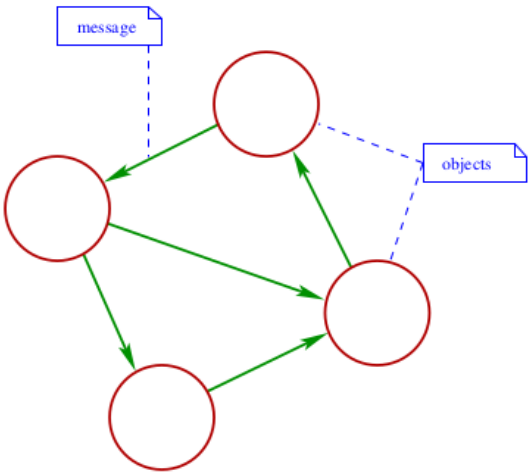


Figure 1: How object interact

It is the principe of **delegating responsibility** of performing an operation

2 Chapter 2: Classes and Methods

2.1 Visibility

In a class definition the distinction between interface and implementation is expressed by **Visibility markers** associated to the class elements :

	Same class	Same package	SubClass	Other
Private	V	X	X	X
Package	V	V	X	X
Protected	V	V	V	X
Public	V	V	V	V

(Final Classes cannot be specialized into subclasses)

2.2 Class and instance Variables

The **variables** defined by a class can belong to two categories

1. Class variables have a value that is shared between all instance of the class (static)
2. Instance variables have a value that is individual to each instance of the class

2.3 Class and Instance Methods

Methods of an object can be classified as follows :

1. Class methods can only access class variables (static)
2. Instance methods are able to read and write all the variable of the object

In its simpliest form, a class declatation is expressed as follows :

[public] [private] class **ClassName**{}

ClassName should be on a file name **ClassName**.java

2.4 Variable

A variable takes the following form :

[visibility] [attributes] type **VariableName**;

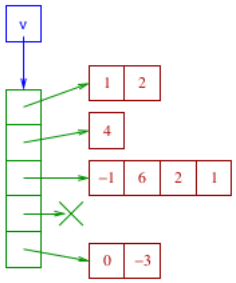
visibility = {private, public}
An *attributes* can be

- static : The variable is a class rather than an instance variable
- final : The value of the variable cannot be modified after it's first assignment
- transient : The value of the variable is not considered to be part of the state of the object
- volatile : The value of the variable can be read or written by mechanisms that are external to the current fragment of code

(A local variable cannot have visibility markers)
In java a **primitive type** represents a storage area suited for a simple value (byte, short, int, char, ...)

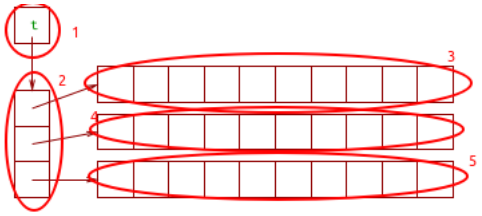
2.5 Array Types

An array variables stores a reference to a vector of data. The components of this vector can either have a primitive or reference type, or be arrays themselves.
Example of declaration : int[ ][ ] v.
The array is dynamic, we don't need to specified the size in the declaration.



We can instantiate an arrays by "**new** *type*[expr]", EXAMPLE :

```
int[][] t = new int[3][10]; //this create 5 objects
```



(Each line can have a different dimension)

2.6 Declaring a Method

[visibility] [attributes] ReturnType **methodName**(type1 param1, type2, param2, ...){...};

The *visibility* marker is identical to the one of variables, but the *attributes* can be :

- STATIC : The method is a class rather than an instance method
- NATIVE : The method is implemented outside the program (could be a different language) the body of this kind of method is replaced by ';'

2.7 Polymorphism

A class may contain several methods that share the same name, provided that the number and/or type of their parameters diffe (signature of the method).

```
class A
{
public void move(int x, int y){ ... }
public void move(float x, float y){ ... }
}
```

3 Chapter 3: Messages, Instantiation, and Initialization of Objects

Sending a message is done **synchronously** : the execution of the invoking method is ssupended while the message is sent.

3.1 Constructors

When an object is created, its instance variables are initialized using the expressions specified in their declaration. Constructor definitions appear at the same place as a method definitions, and take the following form :

```
[visibility] ClassName(type1 param1, ...){ ... }
```

A class can define several constructors, provided that the number and/or type or their parameter differ (constructor polymorphism)
EXAMPLE :

```
 class Point{
private int x, y;

public Point(int x, int y){
  this.x = x;
  this.y = y;
}
public Point(){
  this(0, 0); // call the constructor Point(int x, int y)
}
...
}
```

this(expr1, expr2, ..); is only valid if appears as the first instruction of the caller constructor.

3.2 Garbage collector

Objects are destroyed by means of a **garbage collection** mechanism : The memory allocated to objects that are not reachable anymore by currently active code is automatically freed.
NOTES :

- Garbage collection is generally performed asynchronously (at the same time as program execution). It can however become synchronous in the case of insufficent memory.
- When large objects are not useful anymore to the program, it is a good idea to drop all references to theses objects by using *null*.

There exist a different between **Integer.valueof(10)** and **New Integer(10)**

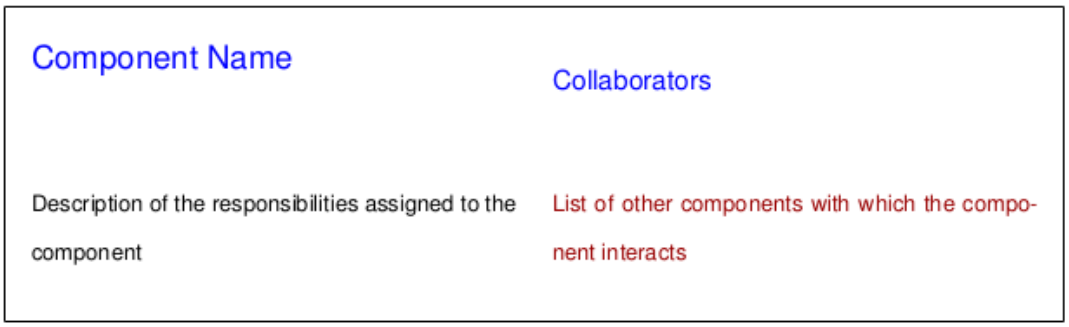
- **valueOf()** give a references to an Object, so done it 100 times give 100 references to one Object.
- **New Integer()** create an new Object, so done it 100 times give 100 Object (not really optimises)

4 Chapter 4: OOP Development Methodology

Developing an OOP amounts to carrying out the following tasks:

1. Specifying a set of usage scenario describing the expected behavior of the system.
2. Defining th components of the program.
3. Assigning to each component a set of responsibilities, and organizing their delegation.

A simple way to do that is using CRC Card :



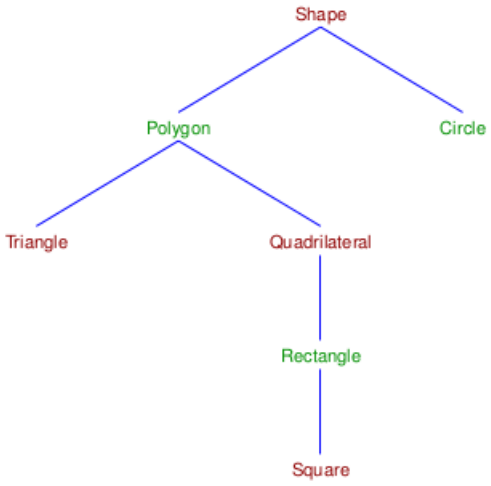
Based on that it will be easier to create Classes and how they will communicate to each other.

5 Chapter 5: Inheritance

5.1 The Class Hierarchy

The classes that that compose an OOP are generally not independent but are linked by a hierarchical relation:

- Descendants of a class (subclasses) are specializations of this class.
- Ancestors of a class (superclasses) correspond to generalization.



5.2 Inheritance

Inheritance is a mechanism related to the hierarchical organization of classes.

- Subclass inherits from the variables and methods of its superclass.
- A subclass is however able to define new variable and methods, as well as to @override inherited elements by its own.
- Inheritance is transitive.

The interface of a subclass necessarily includes the interface of its superclassses. (Btw. every class is a subclass of Object)
For example, if **ClassS** is defined as a direct subclass of ClassG we declare it as follows :

```
class ClassS extends ClassG
{
...
}
```

Then all instances of **ClassS** are able to accept all messages that can be accepted by instances of **ClassG**. **! Multiple inheritance between classes is not allowed in Java !**

5.3 Substitution principle

The inheritance mechanism can be exploited in several ways. A first possibility consists in requiring that subclasses respect all the specification of their superclasses.

Substitution principle : In all situations in which a class can be employed, it should be possible to substitute a subclass without breaking the application.

1. **Specialization**
   - An instance of the subclass represents a particular case of it's superclass instances.
2. **Specification**
   - A class defines behavior that it does not implement itself, but that is meant to be implemented by its subclasses.
3. **Construction**
   - A subclass exploits functionalities implemented by its superclasses, without becoming a subtype of theses classes.
4. **Generalization**
   - A subclass modifies or completely overrides some inherited operations, in order to make them more general.
5. **Extension**
   - A subclass adds new operations to those inherited from its superclasses, without affecting the inherited operations.
6. **Limitation**
   - A subclass restricts the usage of modalities of some inherited operations.
7. **Variation**
   - A subclass and its direct superclass are variants to each other, the direction of the hierarchical relation between them being chosen arbitraly.
8. **Combinaison**
   - A subclass inherits from the elements of more than one direct superclass.

5.4 Abstract Classes

It is sometimes useful to define classes that are not inteded to be instantiated, but that define elements that can be inherited by other classes. Such classes are said to be **abstract**.

An abstract class does not necessarily need to provide a body for each of its methods. An **abstract method** is a method without an implementation, so non-abstract class that inherits it required to implement it.
Defining an abstract class makes it possible

- To share common pieces of implementation between subclasses (non-abstract methods), as well as
- To specify common interface elements between subclassses (abstract methods)

```
abstract classC{
  public abstract int m(int x);
  ...
}

ClassS extends ClassC{
  public int m(int x){ // if m not defined ERROR at compile time
  ...
  }
}
```

5.5 Static/Dynamic link

Object polymorphism can sometimes lead to ambiguous situations.
Example Assume that ClassS and ClassG both defin a method m(), with different implementation, and ClassS inherit from ClassG.

```
ClassG v = new ClassS();
v.m(); // will call ClassS.m()
v.System.Out.print(v.x); // will print ClassG.x
```

- **Instance variables** of an object are accessible by **static link**.
- **Methods** of an object are accessed by **dynamic link**.

1. Static binding happens at compile-time while dynamic happens at runtime.
2. Binding of private, static and final methods always happen at compile time since theses methods cannot be overridden.
3. The binding of overloaded methods is static and binding of overridden methods is dynamic.

Dynamic link need to keep a table with the symbolic link and his memory path, it's ok for instance variable but complicate for methods.

5.6 Accessing Superclass Elements

The **super** keyword in java is a reference variable that is used to refer parent class objects. It can be use

- For variable : **super.v** used if an instance variable of a subclass have the same name that one of his parent class. (variable are static so there is no overriding problems)
- For methods : **super.m()** used for the same reason of the variable BUT methods are dynamics.
- For constructor : **super(x, y)**, it is constructor chaining, for example if ClassS is a subclass of ClassG, and use the constructor super(x, y), the constructor of ClassG is invoked first. this should be the first instruction of ClassS

If ClassG defines a protected element, and ClassS is a subclass of ClassG, the this element can be accessed from ClassS either by the keyword **super**.

5.7 Interface

Recall that in Java, multiple inheritance is forbidden between classes. The language however admits a restricted form of multiple inheritance, corresponding to a unique **specialization** associated with multiple **specification**.

An **interface** is a collection of public method declarations (without implementation). An interface can also define public constants. DEFINED AS FOLLOW

```
interface InterfaceName (extends interface1, interface2,...)
{
    returnType methodName(type param, ...);
    returnType methodName(type param, ...);
    ...
    typeConst constName = constValue;
    ...
}

class ClassName implements interface1 (, interface2,...) ..
{
... // this class have to implements methods declared in interface1,
}
```

6 Chapter 6: Exceptions and Packages

The execution of a program can run into various kinds of unexpected situations : arithmetic overflow, insufficient memory for instantiating object, in/out error, ...
In most OOP such as Java we have **Exceptions** to deal with error.

6.1 Exceptions

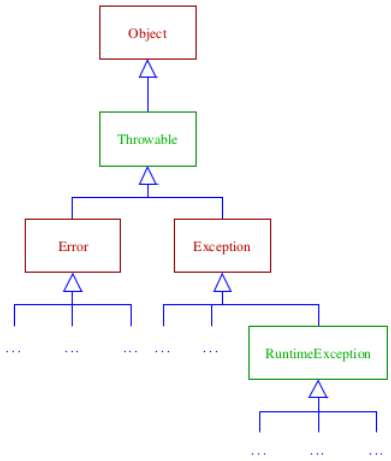
An **Exception** is an object, it is a signal that indicates the occurence of an unexpected situation. Control is then transferred either

- to the invoking method.
- to a dedicated exception handler.

In Java, there are two categories of exceptions:

- **Runtime Exceptions:** Such expression can be triggered essentially everywhere in programs (insufficient memory, attempt to follow an empty reference, ...
- **Checked Exceptions:** They correspond to exceptions that only occur while performing specifical operation : in/our errors, invalid data format during a conversion, ... They are checked at compile time.

All exceptions classes are direct or inderect subclasses of **Throwable**.



In Java we handle expressions by

```
try{
... // executed until an exception is triggered
}catch (ExceptionClass1 e1 (| ExceptionClass2 e2 | ...)){ // (can be omitted)
... // if exception triggered, this block is executed
}
...
finally{ // (can be omitted)
... // Always executed, after the other instructions
}
```

6.2 Delegating Exception

If an exception triggered during the execution of a method is not caught by a catch clause => the method is suspended, and the exception is transmitted to the invoking method.

This delegation of exceptions proceeds from invoked to invoked method, until reaching an appropriate catch clause. (if no such clause found, then the exception is handled by the JVM, which then reports an error to the user.

Method declaration take the following form:

```
returnType methodName(type1 param1, type2 param2, ...) throws Throwable // insteed of
    Throwable we should have a more precise triggered exceptions
{
    if(...)
        throws new Trowable("ERROR MESSAGE');
}
```

Since all exception are subclasses of Throwable, the following code declares a method in which any exception can potentially be triggered without being handled.

The exception classes can be defined as follows :

```
class ExceptionClass extends Exception
{
    public ExceptionClass() {super();}
    public ExceptionClass(String s) {super(s);}
}
```

Example of code:

```
    public Statistics(String fileName) throws StatFileError
    {
        try{
            fis = new FileInputStream(filename);
            isr = new InputStreamReader(fis);
            br = new BufferedReader(isr);
        }catch (IOException e){
            this.close();
            throws new StatFileError("Oppening file.");
        }
    }

    class StatFileError extends Exception
    {
    public StatFileError() {super();}
    public StatFileError(String s) {super(s);}
    }
```

6.3 Packages

Packages are groups of classes sharing the same functional goals. Each package is defined by some number of source files.
Each package is identified by its name, which must be unique among Java developers community.

Such names are composed of a sequence of identifiers, from the most general to the most specific one, seperated by dots '.'
EXAMPLE :

- be.uliege.boigelot.course.ex.eightqueens
- com.google.api.services.analytics

In the repertory it will be represent as : BE/ULIEGE/BOIGELOT/COURSE/EX/EIGHTQUEENS/
The first instruction of a source file mentions the package to which it belongs:

package *packageName;*

To avoid :

```
java.math.BigDecimal v = new java.math.BigDecimal("12");
```

it is preferable to use **Import** such as

import *className*;
import *packageName.**;

so we get :

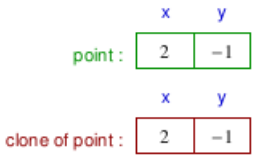
```
import java.math.BigDecimal;
...
BigDecimal v = new BigDecimal("12");
```

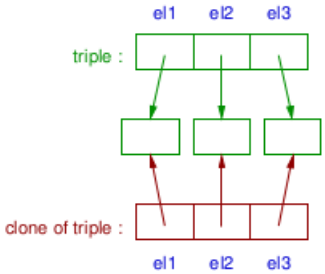
7 Chapter 7: Cloning, Equivalence Checking, and Serialization

In order to duplicate an object that dynamically changes its state, it is not sufficient to copy a reference to this Object.

1. Instantiating a new object from the same class as the object to be copied.
2. Assigning to the instance variables of the new object the same values as thoses of the original object.

It is : Cloning an object. (Here it is **superficial cloning** because if some instance variables reference other objects, then these aux. objects are themselves not cloned)





7.1 Deep Cloning

For application in which a main object maintains exclusive ref. to aux. objects, that are also considered to contain part of its state, one needs to carry out *deep cloning*. Algorithm :

1. Deep cloning recusively the objects ref. by the main object.
2. Cloning superficially the main object.
3. Assigning to each ref. instance variable of the main object clone a ref. to the clone of the corresponding aux. object

7.2 Cloning in java

Cloning an object is performed by sending the message CLONE() to this object. In order to be clonable, an object must implement a method clone() which satisfy :

- **Superficial cloning** invoking the method clone() inherited from Object.
- **Deep cloning** performed by sending clone() messages to the appropriate aux. objects.

Important notes:

- clone() is defined as **protected in Object**, so it's impossible to send clone() message to an obejct whose class doesn't **override** it.
- clone() implemented in Object is only able to clone instances of classes that implement the interface **Cloneable**. (Or CloneNotSupportedException)
- clone() is not equivalent to instantiating an object of the same class using new.
  - Dynamically assigns the instantiation class of the clone to the same class as the cloned object.
  - Creates an object without executing constructors.

```
public class Stack implements Cloneable{
private int nbElem;
private Object[] contents; // need to be cloned
...
public Object clone(){
    Stack copy;
    try {
        copy = (Stack) super.clone();
        copy.contents = (Object[]) this.contents.clone();
    }
    catch (CloneNotSupportedException e) {
        throw new InternalError("Unable to clone");
    }
    return (Object) copy;
    }
}
```

7.3 Equivalence

- '==' ref. to the same object
  - new Integer(3) == new Integer(3) => False
  - ValueOf(3) == ValueOf(3) => True (When big value it might fail)
- 'equals(μ)' method that able to compare object against the object ref. by a parameter μ

Object class defines an **equals** method, this can either be simply inherited by other classes, or been overridden. (by default it is just a '==')
If equals override **hashCode()** method should also be override.

String are immutable, so "String s = str" is ok

7.4 Serialization

Collect all the information that characterizes a given object :

- the class.
- its state (instance variables).
- information about aux classes.

It is **Serialization** which store an object in order to use it in a future execution of the program and to transmit an object from a program to another.

In java, objects can be serialized by invoking the **writeObject** and be reconstructed using **readObject** (from Object(Output/Input)Stream)

- Only instances of classes that implements the **Serializable** interface can be serialized.
- The values of class variables as well as of those declared **transient** are not taken into account.
- Serialization automatically considers all objects that are directly or inderecty ref. by the object being serialized.
- When an object is dezerialized, all objects that is directly or indirectly ref. are deserialized as well, and their mutual ref. are reconstructed.
- To know if an object as the same definition as the one in a different program, it exist **serial number** which is defined automatically by the programming environment when a class implements the Serializable interface (or by the programmer).
  It allow to detect inconsistencies.

PRIVATE STATIC FINAL LONG **serialVersionUID** = *value*;

```
import java.io.*;
public class Stack implements Serializable
{
    private int nbElements;
    private Object[] contents;
    ...
    public Stack(String fileName) throws IOException
    {
        this();
        FileInputStream
        fis = new FileInputStream(fileName);
        ObjectInputStream ois = new ObjectInputStream(fis);
        try
        {
            int n = ((Integer) ois.readObject()).intValue();
            this.nbElements = n;
            for (int i = 0; i < n; i++)
                this.contents[i] = ois.readObject();
        }
        catch (ClassNotFoundException e)
        {
         throw new IOException("Wrong stack format.");
        }
        ois.close();
        fis.close();
    }

    public void save(String fileName) throws IOException
    {
        FileOutputStream
        fos = new FileOutputStream(fileName);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(new Integer(this.nbElements));
        for (int i = 0; i < this.nbElements; i++)
            oos.writeObject(this.contents[i]);
        oos.flush();
        oos.close();
        fos.close();
    }
}
```

8 Chapter 8: Genericity

Using polymorphic object sometimes leads to defining data structures that are too general (using Object)...
And in some situation it can lead to runtime error. (due to casting)
(Object => Int = Compile time error)

8.1 Generic Classes

One augments a class definition with one or many type parameters, that make it possible to specify symbolically.

ILLUSTRATION: The class LinkedList can be turned into a class LinkedList<T>, where T is a parameter.

ADVANTAGES:

- Data structures that only differ in the type of internal or interface elements can share the same code.
- Type checking operation can be carried out at compile time.

It is defined as follow :

[public] [abstract] [final] class ClassName<type1, type2,...>

wheres type1, type2, .. are argument types,

- in variable or method declarations.
- in instantiation expressions.

Argument cannot be primitive !
EXEMPLE:

```
class ClassName<T>{
    private T value;
    ...
    public ClassName(T value){...}

    public T getValue(){return value;}

    public ClassName<T> getClass(){...}
    ....
}

....
public static void main(String[] args){
    ClassName<int> l = new ClassName<int>(10); // ERROR

    ClassName<Integer> l = new ClassName<Integer>(Integer.valueOf(10)); // OK
}
```

8.2 Genericity Restrictions

The genericity mechanism disappears after compilation, after having performed type checking, the compiler removes all information about type parameters.

As a consequence, one cannot program operations that rely on the values of type parameters at runtime. In particular, it is forbidden

- to instantiate object from a type parameter.

```
class C<T>{
    public C(){
    new T(); // Invalid
    }
    ...
}
```

- to create array whose elements are defined with generic type.

```
...
LinkedList<Integer>[] t = new LinkedList<Integer>[10]; //Invalid
...
```

- to define class variables or methods using a type parameters.

```
class C<T>{
    static T x; // Invalid
}
```

- to use generic type with the instanceof operator, or in a type casting operation.

```
...
if (l instanceof LinkedList<Integer>) ... // Invalid
...
```

- to define generic exception types, or catch clauses that rely on type parameters.

```
class C1<T> extends Throwable // Invalid
{...}

Class C2 <T>{
    public void m(){
        ...
        try{
        ...
        }catch(T e){ // Invalid
        ... } } }
```

8.3 Generic Method

Type parameters can also be associated to method declaration (class and instance methods). The scope of such parameters is then limited to their corresponding method.

[visibility] [attributes] <ptype1, ptyp2, ...> returnType methodName(type1 param1, ...)

A message invoking such a method can be sent by evaluating an expression of the form :

reference.<type1, typ2, ...>methodName(expr1, ...)

EXAMPLE:

```
public class ListUtils{
    public static <T> void printElem(LinkedList<T> l, int i){
        System.out.println(l.get(i));
    }
}
...
public static void main(String[] args){
    ...
    ListUtils.<Integer>printElem(l, 2); // l is a linkedList
}
```

8.4 Bounded Type Parameters

In some cases, it is useful to restrict argument types to the subclasses of a given type.

- Declation of a generic class:

  [public] [abstract] [final] class `ClassName<ptype extend Class, ...>`

- Declaration of a generic method:

  `[visibility] [attributes] <ptype extends Class, ...> returnType methodName(Type1 param1, ...)`

EXAMPLE:

```
public class LinkedListOfNumber<T extends Number> // T should be an instance of
    Number
{
private LinkedListOfNumber<T> first;
...
public double sum(){...}
}
```

9 Chapter 9 Concurrency

For some applications, it is however more convenient to develop programs in which several fragments of code are able to run **simultaneously**. Such programs are said to be **concurrent** or parallel.
But it need to face some kinds of problems :

- **Deadlock**, it's a situation which all tasks are waiting for a condition that can only be lifted by the tasks themselves.
- **Livelocks**, it correspond to situations in which tasks repeatedly perform some operations, waiting for a condition to become true.

These problems are impossible to detect at compile time.

9.1 Concurrency in Java

Principe:

- A task corresponds to a **Thread** (represent a unique control flow inside a program)
  - A current control point that precisely identifies the next atomic operation to be executed.
  - A runtime stack used for keeping track of methods that are currently active.
- Objects are stored in a central memory that can be accessed by all threads.
- Copying a value from a central memory to the stack of a thread.
- Operation involving (class or instance) variables are performed on the stack of the corresponding thread.
  - The value of volatile variable is always retrieved from central memory instead of from thread stack. (this value will always be up to date)

9.2 Thread States

- **Initial** The thread has just been created, but has not yet started to execute instructions.
- **Runnable** The thread is able to perform operations.
- **Blocked** The thread es suspended its execution and is waiting for a specific condition to become true in order to resume its operations.
- **Final** The thread has finished its execution.

9.3 Scheduling

The scheduler is reponsible for distributing the processor among those threads.

1. Selecting a runnable thread.
2. Assigning a processor to this thread until either it has executed a sufficient number of instruction, or it becomes blocked.

Notes:

- The scheduler is generally non deterministic. (two execution of a program may lead to different result)
- The scheduler is fair: A thread can never remain indefinitely runnable without being granted a processor.

9.4 Thread Creation

The java language offers two mechanisms for creating a thread.

9.4.1 Instantiating

```
class Task extends Thread{
    public Task(int n){
        super("Task " + Integer.toString(n));
    }
    public void run(){...}
}
...
public static void main(String[] args){
    new Task(1).start();
    new Task(2).start();
}
```

The first one consists in instantiating a class defines as a subclass of a Thread. It must satisfy the follow requierments :

- Its constructor invoke the constructor of Thread that takes a String as a parameter (name of the Thread)
- The class implements a public method void run() that contains the instruction to be executed by the thread.

Instantiating such a class creates a new thread, in the intial state. Sending a message **start()** to the resulting object then makes the thread **runnable**. The thread becomes **final** as soon as the method **run() has finished it's execution**.

9.4.2 Implements

```
class Task implements Runnable
{
    public Task(int n)
    {
        new Thread(this, "Task " + Integer.toString(n)).start();
    }
    public void run()
    {
        ...
    }
}

public class TestTask
{
    public static void main(String[] args)
    {
        new Task(1);
        new Task(2);
        new Task(3);
    }
}
```

- Defining a class that implements the interface **Runnable**. (it require to implement a method "void run()")
- Instantiating the class Thread with two arguments :
  1. a ref. to an instance of the class that implements Runnable
  2. the thread name

9.5 Locks

It is common to have several threads that attempt to simultaneously modify the state of a shared object, by performing non atomic sequences of operations. (This may lead to corrupting this object)

In java, this problem can be solved thanls to the concept of monitor:

- Each object is equipped with a lock that can be acquired and released by threads.
- The execution of a block of instructions can be controlled by the monitor of an object v by using:

```
synchronized(v){...}
```

When executed the current thread:

1. attempts to acquire the lock associated to v. This is only possible provided that no other thread has already acquired this lock. Otherwise, the current thread becomes blocked until it succeeds in acquiring the lock.
2. executes the instructions in the block
3. releases the lock

A method defined with the **synchronized** attribute has its body implicitly included in a synchronized instruction associated to the current object. (if many synchronized methods, they are all using the same monitor, so the same lock of the object)

- Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- Monitor allow a thread to acquire several times the same lock (to avoid deadlock)
- One can also define class methods with the synchronized attribute.

9.6 Thread Synchronization

When several threads exchange data, one sometimes needs to temporally suspend a thread until another has completed some operations.
Synchronization mechanisms

- **Wait()** makes the current thread blocked, and releases the lock associated to the object.
- **notify()** choose a thread waiting for the current object (following wait()) and makes this thread runnable again.
- **notifyAll()** perform a similar operation to notify(), but for all threads that are waiting for the current object.

EXAMPLE:

```
public class Channel{
    private int value;
    private boolean available = false;

    public synchronized int getValue(){
        while(!available){
            try{ wait(); }
            catch (InterruptedException e){ }
        }
        available = false;
        notifyAll();
        return value;
    }

    public synchronized void setValue(int v){
        while(available){
            try{ wait(); }
            catch (InterruptedException e){ }
        }
        available = true;
        value = v;
        notifyAll();
    }
}
```