# Unit 1. Introduction (Part 1)

## Programming Languages, Technologies and Paradigms (LTP)

DSIC, ETSInf

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

etsinf
Escuela Técnica Superior de Ingeniería Informática

# Main goals

- Knowing about the evolution of programming languages and their contributions and impact in the design of other languages.
- Understanding existing (and emerging) programming paradigms (imperative, functional, logic, OO, and concurrent paradigms) and their specific features.
- Understanding different abstraction mechanisms (genericity, inheritance, use of modules) and evaluation strategies (eager/lazy).
- Identifying essential aspects of programming languages: dynamic/static scope, memory management.
- Understanding the criteria for choosing the appropriate programming paradigm/language according to the targeted application, its size, and the programming methodology.
- Understanding the main features of programming languages with regard to the underlying model (paradigm) and main components (type/class system, execution model, abstractions).
- Understanding the concerns of programming language expressivity in the implementation and execution costs.

# The story goes back to 1950...

THE FIFTIES:

- Programmer time was cheap; the machines were expensive:
  *keep the machine busy*
- Sometimes, programs were directly written in machine code. Hand-made compilations were frequent to achieve the best performance for a given hardware:
  *direct connection language-hardware*

NOWADAYS:

- Programmer time is expensive; the machines are cheap:
  *keep the programmer busy*
- Programs are intended to be efficient, but automatically compiled to generate (portable) code which should also be as efficient as possible:
  *direct connection between program and language design: objects, concurrency, etc.*

# Teaching PLs

## Three approaches

1. Programming is a job
2. Programming as a branch of mathematics
3. Programming by concepts

# 1. Programming is a job

- You learn just one paradigm and a single language
- Self-defeating: having a bad experience in, for instance, handling lists in some languages can lead to the erroneous conclusion that managing lists is costly and complicated

# 1. Programming is a job

## ZIP lists in Java

```
class Pair<A, B> {
  private A left;
  private B right;

  public Pair(A left, B right) {
    this.left = left;
    this.right = right;
  }
  public A left() { return left; }
  public B right() { return right; }
  public String toString() {
    return "(" + left + "," +
                 right + ")";
  }
}
public class MyZip {
  public static <A, B> List<Pair<A, B>> zip(List<A> as, List<B> bs) {
  Iterator<A> it1 = as.iterator();
  Iterator<B> it2 = bs.iterator();
  List<Pair<A, B>> result = new ArrayList<>();
  while (it1.hasNext() && it2.hasNext()) {
    result.add(new Pair<A, B>(it1.next(), it2.next()));
  } return result;
}

public static void main(String[] args) {
  List<Integer> x = Arrays.asList(1, 2, 3);
  List<String> y = Arrays.asList("a","b","c");
  List<Pair<Integer,String>> zipped = zip(x, y);
  System.out.println(zipped);
}
```

## Output

```
[(1,a),(2,b),(3,c)]
```

## ZIP lists in Haskell

```
zip :: [a] -> [b] -> [(a,b)]

zip [] xs          = []
zip (x:xs) []      = []
zip (x:xs) (y:ys)  = (x,y):zip xs ys
```

## Use

```
: zip [1,2,3] ["a","b","c"]
[(1,"a"),(2,"b"),(3,"c")]
```

# 2. Programming as a branch of mathematics

- You either learn an ‘ideal’, restricted language (Dijkstra) or else you run out from the real world.

# 2. Programming as a branch of mathematics

Example: formal verification (of a single line (!) program)

## The program

```
while (x<10) x:=x+1;
```

## The proof

One can then prove the following *Hoare triple*:

`{x≤10} while (x<10) x:=x+1 {x=10}`

The condition *C* of the while loop is `x<10`. A useful loop invariant is `x≤10`. Under these assumptions it is possible to prove the following Hoare triple

`{x<10 ∧ x≤10} x:=x+1 {x≤10}`

While this triple can be derived formally from the rules of Floyd-Hoare logic governing the assignment, it is also intuitively justified: *computation starts in a state where* `x<10∧x≤10` *is true, which means simply that* `x<10` *is true. The computation adds 1 to* `x`*, which means that* `x≤10` *is true (for integer x).* Under this premise, the rule for while loops permits the following conclusion:

`{x≤10} while (x<10) x:=x+1 {¬(x<10) ∧ x≤10}`

However, the post-condition `¬(x<10)∧ x≤10` is logically equivalent to `x=10`.

# 3. Programming by concepts

- Learn the semantic concepts and implementation structures leading to a natural description of the languages and their implementations

# 3. Programming by concepts

Features from different 'blocks' may coexist in a given programming language

**Functional language**

(+) Polymorphism
(+) Strategies
(+) Higher-order

**Logic language**

(+) Nondeterminism
(+) Logic variables
(+) Unification

***Kernel* language**

(+) Data abstraction
(+) Recursion
(+) . . .

**Imperative language**

(+) Explicit state
(+) Modularity
(+) Components

**OO Language**

(+) Classes
(+) Inheritance

***Dataflow* language**

(+) Concurrency

# Essential concepts

The following concepts are essential in our presentation of PLs:

- Types and type systems
- Polymorphism
- Reflection
- Parameter passing
- Variable scope
- Memory management

# Types and type systems

A **type** represents the set of values that can be given to a variable or expression. Types:

- Avoid programming **errors**

  *Only programs that make a consistent use of expressions of a given type are **legal***

- Are helpful to give **structure** to the information

  *We think of types as collections of values sharing a number of properties*

- Are helpful to handle **data structures**

  *Types tell us how data structures should be used*

# Types and type systems

## Typed languages

- In **typed** languages all variables have a type (e.g., C, C++, C#, Haskell, Java, Maude).
- **Untyped** languages do not restrict the values adopted by the variables (e.g., Lisp, Prolog).
  - All values have a single (*universal*) type

# Types and type systems

## Typed languages

The type system establishes which kind of values can be associated to variables:

- The type of the **value** and the type of the variable must coincide (e.g., C, Haskell)
- The type of the **value** is *compatible* (according ot the type system) with the type of the variable (e.g., C++, C#, Java).
- Besides, the type of the **value** associated to a given variable may change:
  - Static typing: the type of the value does not change during the execution
  - Dynamic typing: the type of the value may change during the execution

# Types and type systems

## Typed languages

| Typed Languages | = | Program Expressions | + | Type System |
|---|---|---|---|---|

- In languages with **explicit** typing, types are part of the syntax
- In languages with **implicit** typing, types do **not** need to be part of the syntax

# Types and type systems

## Typed languages

- An **untyped** language: Prolog

```
object(key).
object(ball).
thing(X) <- object(X).
```

Variable `X` has no associated type

- A language with **explicit typing**: Java

```
int x;
x = 42;
```

All variables must be declared. Variable declarations include a type

- A language with **implicit typing**: Haskell

```
fac 0 = 1
fac x = x * fac (x-1)
```

The type system automatically infers the type of variable `x`

# Types and type systems

Types are described by means of a *language of type expressions*.

## Example of a language of type expressions:

- Basic or primitive types: `Bool`, `Char`, `Int`, ...
- Type variables: `a`, `b`, `c`, ...
- Type constructors:
  - $\rightarrow$ to define functions,
  - $\times$ for pairing,
  - `[]` to define lists
  - ...
- Rules to build type expressions:

$$\tau \ ::= \ \texttt{Bool} \mid \texttt{Char} \mid \texttt{Int} \mid \cdots \mid \tau \rightarrow \tau \mid \tau \times \tau \mid [\tau]$$

# Types and type systems

## Monomorphic and polymorphic types

- Types whose type expression contains **no** type variable are called monomorphic types or just monotypes
- Types whose type expression contain type variables are called polymorphic types or just polytypes
- A polymorphic type represents an infinite number of monotypes

# Types and type systems

## Examples of type expressions

- Predefined type expressions. Basic types: `Bool`, `Int`, ...

`Bool` is the type of boolean values `True` and `False`

- Functional type expressions.

`Int` → `Int` is the type of function `fact` (see above), which returns the factorial of a number.

- Parametric type expressions.

`[a]` → `Int` is the type of function `length`, which computes the length of a list.

# Types and type systems

## Examples of type expressions

- Predefined type expressions. Basic types: `Bool`, `Int`, ...

`Bool` is the type of boolean values `True` and `False`

monomorphic types

- Functional type expressions.

`Int` $\rightarrow$ `Int` is the type of function `fact` (see above), which returns the factorial of a number.

polymorphic type

- Parametric type expressions.

`[a]` $\rightarrow$ `Int` is the type of function `length`, which computes the length of a list.

type variable

type constructor

# Polymorphism

- It is a feature of programming languages which permits the use of values of different types under a uniform interface
- Both functions and types can be polymorphic:
  - A function can be polymorphic with respect to some of its arguments.

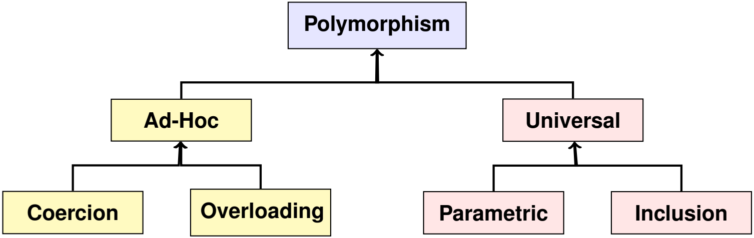
    *a single addition operator (+) can be applied to integers, reals, . . .*

  - A data type can be polymorphic with regard to the types of its components.

    *lists of elements of an arbitrary type*

# Polymorphism

## Different kinds of polymorphism




- Ad-hoc: a finite number of unrelated types is considered
  - Overloading
  - Coercion (type-casting)
- True or Universal: infinitely many types sharing a *common structure* are considered.
  - Parametric polymorphism (genericity)
  - Inclusion polymorphism (inheritance)

# Polymorphism. Overloading

## Ad-Hoc polymorphism: Overloading

- **Overloading**: several functions share a single name (with possibly different implementations).
  - Arithmetic operators $+$, $-$, $*$, $/$, $\ldots$: the monotypes

    ```
    (+) :: Int -> Int -> Int
    (+) :: Float -> Float -> Float
    (+) :: Complex -> Complex -> Complex
    (+) :: Int -> Float -> Float
    ```

    correspond to different uses of $+$
  - The operator $+$ cannot be given a polytype

    ```
    (+) :: a -> a -> a
    ```

    because the meaning of (and how to implement!) the 'sum' of characters, functions, lists, etc., is unclear

# Polymorphism. Overloading

## Example of overloading in Java (1)

In Java, overloaded methods are distinguished by examining the number and types of their parameters

```
/* overloaded methods */

int myAdd(int x,int y, int z){
...
}

double myAdd(double x, double y, double z){
...
}
```

# Polymorphism. Overloading

## Example of overloading in Java (2)

```
public class Overload {
  public void numbers(int x, int y) {
    System.out.println("Method that gets integer numbers");
  }
  public void numbers(double x, double y, double z) {
    System.out.println("Method that gets real numbers");
  }
  public int numbers(String st) {
    System.out.println("The length of "+ st + " is "+
                       st.length());
  }
  public static void main(...) {
    Overload s = new Overload();
    int a = 1;
    int b = 2;
    s.numbers(a,b);
    s.numbers(3.2,5.7,0.0);
    a = s.numbers("Madagascar");
  }
}
```

No match in the number or types of the parameters/result is necessary

# Polymorphism. Coercion

## Ad-Hoc polymorphism: Coercion

- **Coercion**: (implicit or explicit) type conversion of arguments.
- Implicit conversion usually relies on an underlying type hierarchy.

*Most languages provide implicit coercion between integer and real arguments of arithmetic operators*

- Some languages allow for an explicit coercion.

  - *cast* sentence in C-like languages
  - Type transformations in Java:
    - Primitive variables can change their basic type
    - From class to superclass

# Polymorphism. Coercion

## Example of coercion in Java

Implicit conversion in Java:

```
int num1 = 100      // 4 bytes
long num2 = num1    // 8 bytes
```

Explicit conversion in Java:

```
int num1 = 100                     // 4 bytes
short num2 = (short) num1          // 2 bytes
char c = (char) num1               // 2 bytes

String s = Integer.toString(num1)
```

# Polymorphism. Genericity

## Universal polymorphism: Genericity

- **Genericity/Parametric**: the function definition or class declaration has a common structure for a potentially infinite set of types

  - In Haskell we can define and use generic types and functions
  - In Java we can define and use generic classes and methods

# Polymorphism. Genericity

## Example of genericity in Haskell

By using a **generic type** (with type variables), we can define a data structure to represent a *dictionary* and use its entries (of any type):

```
type Entry k v = (k,v)

getKey :: Entry k v -> k
getKey (x,y) = x

getValue :: Entry k v -> v
getValue (x,y) = y
```

With a **generic function** we can compute the length of a list of elements of any type:

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + (length xs)
```

# Polymorphism. Genericity

## Example of genericity in Java (1/2)

We can use a generic class (with parameters) to define an entry of a *dictionary*:

```
public class Entry<K,V>{
  private final K mKey;
  private final V mValue;

  public Entry(K k, V v){
    mKey = k;
    mValue = v;
  }

  public K getKey(){
    return mKey;
  }

  public V getValue(){
    return mValue;
  }
}
```

We can define a **generic method** to compute the length of an array of entries of *any* type:

```
public static <T> int lengthA(T[] inputArray){
   ...
  }
```

# Polymorphism. Genericity

## Example of genericity in Java(2/2)

Use of a dictionary:

parameterization

```
Entry<Integer,String> elem1 = new Entry<>(3,"Programming");
System.out.println(elem1.getValue());
```

Use of a generic method to compute the length of an array

```
Integer[] intArray = {1, 2, 3, 5};
Double[] doubleArray = {1.1, 2.2, 3.3};
System.out.println("Array length =" + lengthA(intArray));
System.out.println("Array length =" + lengthA(doubleArray));
```

# Polymorphism. Genericity

## Some considerations about genericity in Java

- A generic class is a normal one, except that the declaration uses a **type variable** (parameter), that will be instantiated (to a type) when used.
- We can use other generic classes within a generic class
- Generic classes may have several parameters

# Polymorphism. Inclusion

## Universal Polymorphism: Inclusion/Inheritance

- **Inclusion or Inheritance**: the definition of a function is made on types that are related by an inclusion hierarchy.
- In OO Programming, inheritance is the usual mechanism for software reuse and extensibility.

*Classes are organized into a hierarchical structure based on class inheritance*

# Polymorphism. Inclusion

## Universal Polymorphism: Inclusion/Inheritance

IDEA:

If a class B inherits from a class A, then B has the structure and behavior of class A. Besides, we can:

- add new attributes to B
- add new methods to B

Depending on the language, we will be able to:

- redefine inherited methods
- inherit from several classes (a Java class may inherit from a single class, though)

# Polymorphism. Inclusion

## Example of inheritance in Java (1/2)

```java
public class Bicycle {
  protected int cadence;
  protected int gear;
  protected int speed;
  public Bicycle (int startCad, int startSpeed,
                  int startGear){
    cadence = startCad;
    speed = startSpeed;
    gear = startGear;
  }
  public void setCadence(int newValue){
    cadence = newValue; }
  public void setGear(int newValue) {
    gear = newValue; }
  public void applyBrake(int decrement) {
    speed -= decrement;}
  public void speedUp(int increment) {
    speed += increment; }
}
```

# Polymorphism. Inclusion

## Example of inheritance in Java (2/2)

```
public class MountainBike extends Bicycle {
  private int seatHeight;
  public MountainBike(int startHeight, int startCad,
                      int startSpeed, int startGear){
    super(startCad, startSpeed, startGear);
    seatHeight = startHeight;
  }
}
```

- Subclasses are introduced using the keyword **extends**
- New attributes and methods can be added. Methods can be redefined as well.

# Polymorphism. Inclusion

## Example of inheritance in Java (2/2)

```
public class MountainBike extends Bicycle {
  private int seatHeight;
  public MountainBike(int startHeight, int startCad,
                      int startSpeed, int startGear){
    super(startCad, startSpeed, startGear);
    seatHeight = startHeight;
  }
}
```

call to the superclass constructor

- Subclasses are introduced using the keyword **extends**
- New attributes and methods can be added. Methods can be redefined as well.

# Polymorphism. Inclusion

## About inheritance in Java (1/2)

- Class `Object` is the topmost class in any Java hierarchy
- A class which is declared to be `final` cannot specialize into any subclass
- Only simple inheritance is allowed in Java
- Superclass variables may be instantiated to objects of a subclass, but **not** vice versa

### Example of a valid assignment

```
Bicycle b;
MountainBike m = new MountainBike(75,90,25,8);
b = m
```

# Polymorphism. Inclusion

## About inheritance in Java (2/2)

- In Java qualifiers preceding attributes and methods are used to establish visibility of instance variables and methods from a class
  - **Private**: no visible from subclasses or other classes.
    *In order to read or write private attributes, appropriate methods must be provided*
  - **Protected**: visible from all subclasses and all classes in the same package.
  - **Public**: no restrictions. Public members are visible from any other class.
  - **Default**: Default members are visible from any other class in the same package.

# Polymorphism. Inclusion

## About inheritance in Java (2/2)

| | Class | Package | Subclass | Other |
|---|---|---|---|---|
| Public | Yes | Yes | Yes | Yes |
| Private | Yes | No | No | No |
| Protected | Yes | Yes | Yes | No |
| Default | Yes | Yes | No | No |

Cuadro: Visibility in Java

# Polymorphism. Inclusion

## Example: redefining inherited methods in Java (1/2)

```
pupublic class Employee {
  protected String name;
  protected int nEmployee, salary;
  static private int counter = 0;
  public Employee (String name, int salary){
    this.name = name;
    this.salary = salary;
    nEmployee = ++counter;
  }
  public void increaseSalary(int wageRaise){
    salary += (int) (salary*wageRaise/100);
  }
  public String toString(){
    return "Num. Employee " + nEmployee +
           " Name: " + name + " Salary: " + salary;
  }
}
```

# Polymorphism. Inclusion

## Example: redefining inherited methods in Java (2/2)

```
public class Executive extends Employee{
  protected int budget;
  void assignBudget(int b){
       budget = b;
  }
  public String toString(){
    String s = super.toString();
    s = s + " Budget: " + budget;
    return s;
  }
}
```

Example of use:

```
Executive boss = new Executive("Thomas Turner", 1000);
boss.assignBudget(1500);
boss.increaseSalary(5);
```

# Polymorphism. Inclusion

## Inheritance in Java: Abstract classes

- An abstract class is one which is declared to be `abstract`
  - Classes containing an `abstract` method must be `abstract`.
  - `abstract` methods have no implementation.
  - Abstract classes cannot be instantiated.
- Non-abstract subclasses of abstract classes must implement each inherited abstract method

# Polymorphism. Inclusion

## Example: use of abstract classes in Java (1/2)

```
public abstract class Shape {
  private double x, y; // Position of the shape
  public Shape (double initX, double initY){
    x = initX; y = initY;
  }
  public void move(double incX, double incY){
    x = x+incX; y = y+incY;
  }
  public double getX(){ return x; }
  public double getY(){ return y; }
  public abstract double perimeter();
  public abstract double area();
}
```

# Polymorphism. Inclusion

## Example: use of abstract classes in Java (2/2)

```java
public class Square extends Shape {
  private double side;
  public Square (double initX, double initY, double initSide){
    super(initX,initY); // Call to superclass constructor
    side = initSide;
  }
  public double perimeter(){ return 4*side; }
  public double area(){ return side*side; }
}

public class Circle extends Shape {
  private double radius;
  public Circle(double initX, double initY, double initRadius){
    super(initX,initY); // Call to superclass constructor
    radius = initRadius;
  }
  public double perimeter(){ return 2*Math.Pi*radius; }
  public double area(){ return Math.Pi*radius*radius; }
}
```

# Polymorphism. Inclusion

## Example: use of abstract classes in Java (2/2)

```
public class Square extends Shape {
  private double side;
  public Square (double initX, double initY, double initSide){
    super(initX,initY); // Call to superclass constructor
    side = initSide;
  }
  public double perimeter(){ return 4*side; }
  public double area(){ return side*side; }
}

public class Circle extends Shape {
  private double radius;
  public Circle(double initX, double initY, double initRadius){
    super(initX,initY
    radius = initRad
  }
  public double perimeter(){ return 2*Math.Pi*radius; }
  public double area(){ return Math.Pi*radius*radius; }
}
```

How do we extend the example to deal with a new form (e.g., triangles)?

# Polymorphism. Inclusion

## Inheritance in Java: Interfaces

An interface declares attributes and operations to be defined in classes implementing (implements) such an interface

```
public interface MyInterface {
  public int method1(...);
  ...}

public class MyClass implements MyInterface {
  public int method1(...) {...}
...}
```

- Interface methods can be abstract, static, and default (they include a default implementation). Their visibility can be "`public`", "`private`" and "default", but not "`protected`".
- Attributes are static and final (i.e., constant)
- Classes are allowed to implement several interfaces
- Interfaces are allowed to inherit from other interfaces (extends)

# Question: Inclusion and genericity

Consider the following class definitions:

```
class Shape { /*...*/ }
class Circle extends Shape { /*...*/}
class Rectangle extends Shape { /*...*/ }
class Node<T> { /*...*/ }
```

Is there any compilation error in the following code? Why?

```
Node<Circle> nc = new Node<Circle>();
Node<Shape> ns = nc;
```

# Question: Inclusion and genericity

Answer the following questions:

- Can an interface inherit from a class?
- Can we obtain instances from an interface?

# Reflection

## What is reflection

When looking yourself at the mirror, you may:

- See your reflected image and
- React according to what you see

- In programming languages, reflection enables a program to:
  - see its own structure and
  - manipulate its own code
- LISP was the first language with reflection; it is also present in several modern programming languages (Java, C#, JavaScript, PHP, Perl, etc).

With reflection we can write programs that monitor their own execution; eventually a program can introduce runtime (self)modifications to dynamically adapt its behavior to the environment

# Reflection

## Languages with reflection

- Languages with reflection treat their own instructions as values of an specific datatype which is a first-class type of the language (languages without reflection just see strings).
- Reflection can be seen as the ability of a language to be its own metalanguage

  > *We call metalanguage to the language which is used to write metaprograms (programs dealing with other programs: compilers, analizers, etc.)*

# Reflection

## Use with caution

- A bad use of reflection
  - may hinder performance

    *if you can do it without reflection, do not use it*
  - may compromise security by showing up unexpected details of the code in certain contexts

    *reflection breaks abstraction, and private attributes and methods can be reached*
- It is an advanced but simple feature, especially in functional languages due to the natural data/program duality of these languages (homoiconicity).

# Reflection

## Reflection in Java

- Reflection is available in Java through an specific library (java.lang.reflect)

The library provides classes for the structured representation of information about classes, variables, methods, etc.

- Reflection in Java permits the runtime inspection of classes, interfaces, attributes and methods without knowing the names of the interfaces, attributes and methods at compile time. In this way, we can

- Read a `String` from the keyboard and use it to create an object and give it a name, or call a method with this name
- reading all getter methods (get) or modifiers (set) of a class
- accessing private fields and methods of a class.

# Reflection

## Example: use of reflection in Java

```
import java.lang.reflect.*;
public class MyClass {...}

...
Class myClassObj = new MyClass();
// get the class information:
Class<? extends MyClass> objMyClassInfo =
                              myClassObj.getClass();
// get the fields:
Field[] allDeclaredVars = objMyClassInfo.getDeclaredFields();
// travel the fields:
for (Field variable : allDeclaredVars) {
    System.out.println("Name of GLOBAL VARIABLE: " +
                          variable.getName);
}
```

Further methods defined in `Class`:

```
Constructor[] getConstructors();
Field[]       getDeclaredFields();
Method[]      getDeclaredMethods();
...
```

# Procedures and control flow

Some important concepts in programming languages concern the control flow of program execution, and how calls to functions and procedures are processed.

- **Parameter passing**. When a method or function is called there is a change in the execution context which can be done in different ways. We discuss the main ones.
- **Variable scope**. We need to know whether an object or variable can be reached from a given program point. This can be done statically or dinamically.

# Parameter passing

The use of functions, methods and procedures is an essential abstraction mechanism in program development. They solve specific tasks which are executed after a call with an appropriate parameterization.

- CALL: $f(e_1, \ldots, e_n)$ with $e_1, \ldots, e_n$ being expressions.
  - when the call is executed, the control flow moves to the body of function $f$. After completion, the control flow returns ‘below’ the program point issuing the call
  - $e_1, \ldots, e_n$ are called input or actual parameters
- DECLARATION: $f(x_1, \ldots, x_n)$ with $x_1, \ldots, x_n$ being variables.
  - $x_1, \ldots, x_n$ are called formal parameters
  - Formal parameters are variables which are local to the body of function $f$ (i.e., reachable within the body of $f$ only)

# Parameter passing

A number of parameter passing mechanisms can be considered:

- Call by value
- Call by reference/call by address
- Call by need

There are other parameter passing mechanisms, but these are the most frequently used in programming languages

# Parameter passing

## Call by value

The values $v_i$ of the input parameters $e_i$ are computed and then copied into the formal parameters $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);
```

# Parameter passing

## Call by value

The values $v_i$ of the input parameters $e_i$ are computed and then copied into the formal parameters $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);
```

a = 10

# Parameter passing

## Call by value

The values $v_i$ of the input parameters $e_i$ are computed and then copied into the formal parameters $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);                          a = 10
```

**Function call:**

Value `10` is copied into the formal parameter `v`

# Parameter passing

## Call by value

The values $v_i$ of the input parameters $e_i$ are computed and then copied into the formal parameters $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{                                   v = 10
      v = v + v;
}
...
int a = 10;
inc(a);                             a = 10
```

# Parameter passing

## Call by value

The values $v_i$ of the input parameters $e_i$ are computed and then copied into the formal parameters $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
      v = v + v;
}                                     v = 20
...
int a = 10;
inc(a);                               a = 10
```

# Parameter passing

## Call by value

The values $v_i$ of the input parameters $e_i$ are computed and then copied into the formal parameters $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);                         a = 10
```

- Variable `a` is NOT modified: a working copy is used inside the body of function `inc`.

# Parameter passing

## Call by reference

A reference to the memory is passed so that the body of function $f$ works on the same memory object

- Input parameters $e_i$ which are **not** variables are treated as in call by value
- In contrast, if $e_i$ is a variable (e.g., $y_i$), then any assigment to the formal parameter $x_i$ in the body of $f$ also modifies the value associated to variable $y_i$

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);
```

# Parameter passing

## Call by reference

A reference to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
     v = v + v;
}
...
int a = 10;                    a = 10
inc(a);
```

# Parameter passing

## Call by reference

A reference to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);
```

$a = 10$

## Function call:

The formal parameter `v` is given the memory address of `a`

# Parameter passing

## Call by reference

A reference to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{                                  v = 10
      v = v + v;
}
...
int a = 10;
inc(a);                            a = 10
```

# Parameter passing

## Call by reference

A reference to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
      v = v + v;
}                                   v = 20
...
int a = 10;
inc(a);                             a = 20
```

# Parameter passing

## Call by reference

A reference to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a);                    a = 20
```

- Variable *a* IS modified: the working memory area for `v`Â and `a` is the same

# Parameter passing

## Call by need

- Expresions from the input parameters are evaluated only when used in the body of the function
- Used in some functional languages

### Example

Consider the following function that returns the double of the second argument

```
sel2nd x y = 2*y
```

With call-by-need, a call `sel2nd (2*3) 5` would not evaluate the value of expression `2*3` as it is not used in the expression defining the function.

# Parameter passing

## Some remarks

- In call by value, input expressions $e_i$ are first evaluated; then, the value $v_i$ is copied into the local variable $x_i$ (formal parameter). This is in contrast to call by need.
- In call by reference, input expressions $e_i$ which are not variables are also evaluated and the computed value $v_i$ is also copied into the formal parameter $x_i$.

# Variable scope (1/2)

- A variable is a *name* referring to a memory address
- During the execution of the program, the variables, functions, constants, etc., which are identified by the names in the program can be reachable, or not
- The scope of a name is the code portion where it is visible (its value can be read/written).

# Variable scope (2/2)

- The instant when the binding (association) is established is called the link time
  - With static scope, it is defined in *compilation time*
  - With dynamic scope, it is defined *during the execution*
- Modern programming languages use static scope.
- Each programming language fixes its particular variable scope approach
  - Java uses `private`, `public`, and `protected` attribiutes and also the class/packages hierarchy system.

# Variable scope

## Example: computing the variable scope (1/2)

```
program scope;
type
  TArray : array [1..3]
               of integer;
var
  a: TArray;
procedure one;
  procedure swap(
               i,j:integer);
  var aux : integer;
  begin {* swap *}
               aux := a[i];
               a[i] := a[j];
               a[j] := aux;
  end {* swap *};
  procedure two;
    a : TArray;
  begin {* two *}
    a[1] := 1;
    a[2] := 2;
    a[3] := 3;
    swap(1, 2);
    writeln(a[1],' ',a[2],
       ' ',a[3]);
  end {* two *};
begin {* one *}
  a[1] := 0;
  a[2] := 0;
  a[3] := 0;
  two;
end {* one *};
begin {* scope *}
  one;
end. {* scope *}
```

# Variable scope

## Example: computing the variable scope (1/2)

```
program scope;
type
  TArray : array [1..3]
                of integer;
var
  a: TArray;
procedure one;
  procedure swap(
                i,j:integer);
  var aux : integer;
  begin {* swap *}
                aux := a[i];
                a[i] := a[j];
                a[j] := aux;
  end {* swap *};
  procedure two;
    a : TArray;
  begin {* two *}
    a[1] := 1;
    a[2] := 2;
    a[3] := 3;
    swap(1, 2);
    writeln(a[1],' ',a[2],
       ' ',a[3]);
  end {* two *};
begin {* one *}
  a[1] := 0;
  a[2] := 0;
  a[3] := 0;
  two;
end {* one *};
begin {* scope *}
  one;
end. {* scope *}
```

Which values are stored in array `a` at the end of the execution?
What does `writeln` print on the screen?

# Variable scope

## Example: computing the variable scope (2/2)

### Assuming static scope...

Which values are stored in array `a` at the end of the execution? What does `writeln` print on the screen?

In **compilation time**:

- Lines 27 to 30 (body of function `one`) refer to variable `a` which is bound to the global variable in line 6 (note that `one` has no local declaration for any variable)
- Lines 19 to 24 (body of function `two`) refer to a variable `a` which is bound to the local variable of `two` defined in line 17: local variables hide global variables with the same name
- Lines 12 to 14 (body of function `swap`) refer to a variable `a` which is bound to the global variable in line 6: like `two`, procedure `swap` is local to `one`

# Variable scope

## Example: computing the variable scope (2/2)

### Assuming static scope...

Which values are stored in array `a` at the end of the execution? What does `writeln` print on the screen?

Therefore:

- The main program calls to procedure `one` (line 33).
- The values in the global array are initialized to 0, 0 and 0 (lines 27 to 29)
- A call to `two` initializes a local array to 1, 2 and 3; the global array does not change (lines 19 to 21)
- The call to `swap` changes the values of the global array to 0, 0, and 0. The local array of `two` does not change.
- The values of the local array (1, 2 and 3) are printed

# Memory management

Concerns the different methods and procedures which are used to maximize the memory use.

## Impact in the design of the programming language

Some design choices can only be explained by the desire of the language designers to use a particular memory management technique.

# Memory management

## The need of memory management

Some program and data elements requiring storage during program execution

- The code of the compiled program
- Temporaries in expression evaluation (e.g., $(x + y) \times (u + v)$) and in parameter passing (e.g. when a subprogram is called, a list of actual parameters must be evaluated and the resulting values stored until the evaluation of the entire list is complete)
- Subprogram call and return operations
- Input-output *Buffers*
- Data structure insertion and destruction operations (e.g. `new` in Java or `dispose` in Pascal)
- Component insertion and deletion operations in data structures (e.g. the Perl `push` function adds an element to an array)

# Memory management

## Memory management approaches

Main kinds of memory management approaches:

### Static allocation

Computed and assigned in compilation time

- Efficient but incompatible with recursion or dynamic data structures

### Dynamic allocation

Computed and assigned during the execution

- stack-based
- heap-based

# Memory management

## Static allocation

Memory allocation computed in compilation time. It remains fixed throughout the execution.
It can be used with:

- Global variables
- Program's machine language translation
- Variables that are local to a single routine but retain their values from one invocation to the next
- Numeric and string-valued constant literals
- Tables produced by compilers that are used by run-time support routines for debugging, dynamic-type checking, . . .

# Memory management

## Static allocation

Memory allocation computed in compilation time. It remains fixed throughout the execution.
It can be used with:

- Global variables
- Program's machine language translation
- Variables that are local to a single routine but retain their values from one invocation to the next
- Numeric and string-valued constant literals
- Tables produced by compilers that are used by run-time support routines for debugging, dynamic-type checking, . . .

Efficient but incompatible with recursive subprogram calls and data structures whose size depends on computed or input data

# Memory management

## Dynamic allocation using a stack

The simplest run-time storage-management technique to handle the activation record in function calls during the program execution (a pointer to the top of the stack suffices)

### Stack-based allocation

- Free storage when the execution starts is set up as a sequential block in memory (a stack)
- As storage is allocated, it is taken from sequential allocations in this stack block beginning at one end
- Storage must be freed in the reverse order of allocations so that a block of storage being freed is always at the top of the stack

# Memory management

## Dynamic allocation using a *heap*

A heap is a region of storage in which subblocks can be allocated and deacollated at *arbitrary times*

- This kind of storage management is needed when the language allows for data structures (like sets or lists) whose size may change during the execution
- The allocated elements are always of the same fixed size, or of variable size
- Deallocation is
  - explicit (e.g. C, C++, Pascal)
  - implicit (when it is no longer possible to reach the allocated element from any program variable)

# Memory management

## Dynamic allocation using a *heap*

A heap is a region of storage in which subblocks can be allocated and deacollated at *arbitrary times*

- This kind of storage management is needed when the language allows for data structures (like sets or lists) whose size may change during the execution
- The allocated elements are always of the same fixed size, or of variable size
- Deallocation is
  - explicit (e.g. C, C++, Pascal)
  - implicit (when it is no longer possible to reach the allocated element from any program variable)
- *Garbage collector*: language mechanism that identifies unreachable elements and returns them to the free-space

# Unit 1. Introduction (Part 2)

## Programming Languages, Technologies and Paradigms (LTP)

DSIC, ETSInf

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

etsinf
Escuela Técnica Superior de Ingeniería Informática

# Programming Paradigms

## Key factors for a successful PL

- Expressive power: write clear, compact and maintainable source code
- Easy to learn
- Portable and safe
- Multi-platform and furnished with appropriate development tools and environments
- Financial support
- The migration from applications written in other languages is not difficult (e.g., C++ $\rightarrow$ Java)
- Multiple libraries are available for a variety of applications
- Downloading open code written in the language is possible

# Programming Paradigms

## Definition of programming paradigm

Basic model for designing and developing programs which provides methods and techniques for producing programs according to specific guidelines (style and approach to solve a given problem)

Main paradigms:

- Imperative
- Declarative
  - functional
  - logic
- Object-oriented
- Concurrent

There also are the so-called *emerging* paradigms

# Imperative Paradigm

A program is considered as a sequence of commands that changes the state of a machine

- It establishes how to proceed $\rightarrow$ algorithm
- The main concept is the machine state, which is given by the values of the variables stored in the memory
- Instructions are sequentially processed and the program builds the sequence of machine states leading to the solution
- This model strongly follows the usual machine architecture (*Von Neumann*'s)
- Programs are structured in blocks and modules.
- Efficient, difficult to modify and verify, with side effects

# Imperative Paradigm

## Example: Pascal

Function length in Pascal:

```
function length (l : list): integer
var
   b : boolean;
   aux : list;
begin
   b := is_empty(l);
   case b of
     true : length := 0;
     false : begin
                aux := tail(l);
                length := 1+length(aux);
              end;
   end;
end
```

# Imperative Paradigm

## Side effects

Two calls to the same function with the same arguments may return different results

```
program test;
var
   flag : boolean;
function f (x : integer) : integer;
   begin
     flag := not flag;
     if flag then f := x else f := x+1;
   end;
begin
   flag := false;
   write(f(1));
   write(f(1));
end
```

global variable

f changes the value of the global variable

# Imperative Paradigm

## Side effects

Two calls to the same function with the same arguments may return different results

```
program test;
var
   flag : boolean;
function f (x : integer) : integer;
   begin
     flag := not flag;
     if flag then f := x else f := x+1;
   end;
begin
   flag := false;
   write(f(1));
   write(f(1));
end
```

Program outcome:

```
> test
1
2
```

# Imperative Programming

## Features

- The main point is how to solve a problem
- The execution order crucially depends on the sequence of program statements
- Destructive assignment (new values given to a variable destroy any previously associated value) $\rightarrow$ understanding the code is harder (even more when there are side effects)
- The programmer is responsible for all control issues
- More complex than usually admitted (as witnessed by the complex semantic definitions or the difficulty of the associated techniques, e.g., formal verification techniques)
- Parallelization is difficult
- Programmers often prefer to neglect some advanced and good features in exchange for a faster execution

# Declarative Paradigm

A program describes the properties of the desired solution. The algorithm (set of instructions) which is used to find a solution is not specified

- Kowalski's insight:

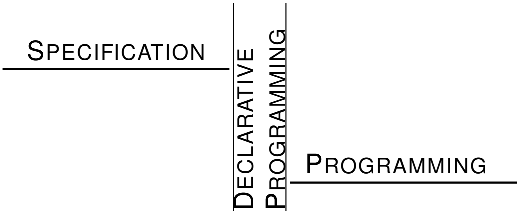
*PROGRAM = LOGIC + CONTROL*

  - Logic: is about the what's
  - Control: is about the how's
- The programmer focuses on the logic aspects of the solution. Control aspects are left to the compiler/system
- Easy to verify and modify; clear and concise

# Declarative Paradigm

Declarative programs can be thought of as **executable specificacions**.

Declarative language = (executable) SPECIFICATION language
(high-level) PROGRAMMING language



# Declarative Paradigm

## Specification vs. programming

### Specification: Definition of a mathematical function

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1)+fib(n-2)
```

# Declarative Paradigm

## Specification vs. programming

Specification: Definition of a mathematical function

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1)+fib(n-2)
```

Program (two versions):

The specification!

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1)+
          fib(n-2)
```

Optimized (with accumulator)

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib_aux(1,1,n)
fib_aux(x,y,0)  = x
fib_aux(x,y,n)  =
       fib_aux(y,x+y,n-1)
```

# Declarative Paradigm

- **Functional Paradigm** (based on $\lambda$-calculus)
  - Data structures and functions for manipulating them are defined by means of equations (s=t)
  - polymorphism
  - higher-order
- **Logic Paradigm** (based on first-order logic)
  - relations among objects are defined by means of rules:

    *If C1 and C2 and $\cdots$ and Cn, then A,*
    written $A \leftarrow C1, C2, \ldots, Cn$

  - logic variables
  - indeterminism

# Declarative Paradigm

## Example: Haskell and Prolog

Function `length` on lists:

### Haskell

```
data list a = [] | a:list a

length [] = 0
length (x:xs) = (length xs) + 1
```

### Prolog

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M + 1.
```

# Declarative Paradigm

## Features

- Specification of what is a solution to a given problem
- The order of program sentences does not change the program semantics
- Expressions denote values that do not depend on the program context (referential transparency)
- High level programming:
  - simpler semantics
  - automatic control
  - amenable for parallelization
  - simpler maintenance
  - more expressive
  - smaller code
  - more productivity
- Efficiency: comparable to imperative languages like Java
- Faster acquisition of programming skills
- Some features of real systems are difficult to model in the declarative setting

# Declarative *vs* imperative paradigms

## Imperative Paradigm

| | |
|---|---|
| PROGRAM | Transcript of an algorithm |
| INSTRUCTIONS | Control of an underlying machine |
| COMPUTATIONAL MODEL | State machine |
| VARIABLES | References to machine memory |

# Declarative *vs* imperative paradigms

## Declarative Paradigm

LOGIC
as a programming language

| | |
|---|---|
| PROGRAM | Specification of a problem |
| INSTRUCTIONS | Logic Formulas |
| COMPUTATIONAL MODEL | Inference machine |
| VARIABLES | Logic variables |

# Declarative *vs* imperative paradigms

## Example

### What is the purpose of this imperative program?

```
void f(int a[], int lo, hi){
  int h, l, p, t;

  if (lo<hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l<h)&&
             (a[l] <= p))
        l = l+1;
      while ((h>l)&&
             (a[h] >= p))
        h = h-1;
      if (l<h) {
        t = a[l];
        a[l] = a[h];
        a[h] = t;
      }
    }

    a[hi] = a[l];
    a[l] = p;

    f(a, lo, l-1);
    f(a, l+1, hi);
}
```

# Declarative *vs* imperative paradigms

## Example

### What is the purpose of this declarative program?

```
f :: Ord a => [a] -> [a]

f [] = []
f (p:xs) = (f lesser) ++ [p] ++ (f greater)
          where
            lesser = filter (< p) xs
            greater = filter (>= p) xs
```

# Declarative *vs* imperative paradigms

## Example

### What is the purpose of this declarative program?

```
f :: Ord a => [a] -> [a]

f [] = []
f (p:xs) = (f lesser) ++ [p] ++ (f greater)
          where
            lesser = filter (< p) xs
            greater = filter (>= p) xs
```

- No variable assignment
- No indices to an array
- No memory management

# Object-Oriented Paradigm

**Embed state and operations into objects**

- Object: state + operations
- Important concepts: *class*, *instance*, *subclass*, inheritance
- Essential elements:
  - abstraction
  - encapsulation
  - modularity
  - hierarchy

# Object-Oriented Paradigm

## Example: Java

Class *Circle* is an abstraction of the notion of *circle*:

```
public class Circle {      // class name
  double radius;           // variables (state)
  String color;

  double getRadius() {...} // methods (operations)
  double getArea() {...}
}
```

Classes are used to define instances representing specific objects (circles with a particular radius and color)

```
Circle c1, c2;
c1 = new Circle(2.0, "blue");
c2 = new Circle(3.0, "red");

Circle c3 = new Circle(1.5, "red");
```

# Concurrent Paradigm

- Concurrent programming languages are used to program the simultaneous execution of multiple interactive tasks
- Such tasks consist of a set of processes created by a single program:

> Concurrent access to databases, use of the resources provided by an operating system, etc.

- Concurrent programming began with the introduction of interruptions in the late fifties.
  - Interruption: a hardware mechanism to break the execution flow of a program in such a way that the CPU transfers the control to a given address, where a special routine (or handler) performs the appropriate actions that are associated to the interruption

# Concurrent Paradigm

## Problems associated to concurrency

- Corruption of shared data

  For instance, if two programs concurrently write on the same printer, the output can be unreadable

- Deadlock among processes sharing resources

  Process A demands shared resources R1 and R2. In order to avoid the aforementioned problem, A tries to lock them by first requiring R1 and then R2. Simultaneously, process B tries to lock R2 and then R1. Both processes get only one of the resources and wait forever for the other

- Starvation of processes never obtaining a given resource.

  The OS enqueues the processes trying to gain access to a shared resource according to their priority. Less prioritary processes may fail to obtain resources demanded by high-priority processes.

- Indeterminism in the coordination of actions from different processes.

  Debugging concurrent programs can be very difficult due to such dependencies

# Concurrent Paradigm

## Main concepts: First abstractions (1/2)

- First attempts to define concurrent languages just added OS-supported primitives to launch processes (coroutines) as part of the execution of programs written in a sequential language (Simula).
  - Problem: low level and lack of portability
- Dijkstra introduced the first abstractions (1965-71).
  - **Concurrent Program:** a set of asynchronous sequential processes making no assumption about the relative progress of other processes
  - semaphores were introduced as a synchronization mechanism

# Concurrent Paradigm

## Main concepts: First abstractions (2/2)

- Hoare additionally introduced the notion of critical region to avoid deadlocks
  - managing critical region was costly and modularity was difficult to achieve
- In 1974 a new approach to encapsulate shared resources was introduced: monitors, inspired by sequential programming ADTs.
  - The first high-level concurrent language with monitors was concurrent Pascal (1975), subsequently incorporated into Modula-2.
- New architecture independent models (CSP, CCS, $\pi$-calculus, Petri nets, PVM) were introduced for the analysis of concurrent programs
  - New constructs inspired in such models were introduced in a number of languages. For instance, CSP inspired Occam's channels and Ada's remote calls.

# Concurrent Paradigm

## Example: threads in Java (1/2)

Two ways to create threads in Java:

- using inheritance (`extends`)
- using interfaces (`implements`)

### Inheritance

Define a subclass `MyThread` of the Java class `Thread`

```
class MyThread extends Thread {
      public void run () {
        // encode here the task to be executed
        // by the thread
      }
}
```

# Concurrent Paradigm

## Example: threads in Java (2/2)

Create and use an instance of `MyThread`

```
MyThread t1 = new MyThread();
t1.setPriority(5)
t1.start();
System.out.println("Now, I can do other things");
// ...
```

- Method `start` initiates the execution of the thread (with a call to `run`)
- The priority assignment is optional
- The message is displayed disregarding the execution of the thread.

# Concurrent Paradigm

## About concurrency in Java

- Java class Thread provides a built-in support of concurrent programming (without additional libraries).
- Threads are similar to processes, although all resources used by threads belong to a 'root' program.
  - In contrast, processes may have their own memory addresses and execution environment.
- There are specific functions to create (and run, suspend, resume, abort, priorize, sinchronize, etc) such threads
- The JVM is able to organize them; however, avoiding undesirable behaviors (deadlock, starvation, etc.) is left to the programmer.
- The implementation of concurrent communication relies on the use of shared memory. Accordingly, some locking mechanism must be used to coordinate the threads. Actually, each object is implicitly locked if some thread is using it.

# Parallel programming

## Main goal:

accelerating time-consuming algorithms by splitting the execution time as much as possible, by using several processors to distribute the data and the execution workload.

- After the introduction of microprocessors in 1975, processes began to be concurrently executed in different processors. In this way, the implicit assumption (essential for monitors and semaphores) of a common memory where shared variables would be placed became unfeasible.
  - New process communication approaches were proposed. For instance, message passing approaches among processors (rendez-vous).
- Early parallel lenguajes were sequential languages (Fortran, C) extended with proprietary message passing libraries.

# Parallel *vs* Concurrent Programming

| | PARALLEL | CONCURRENT |
|---|---|---|
| GOAL | Efficiency: workload *distribution* | Interaction: *simultaneous* processes |
| PROCESSORS | more than one | one or more |
| COMMUNICATION | message exchange | shared memory |

# Interaction-based Paradigm

- The traditional paradigm follows Von Neumann's *programming as calculation* style.
  - a program describes the sequence of steps that are necessary to yield a result out from the program inputs
- This model does not fit the requirements of some areas: HCI, robotics, software agents, AI, service oriented applications, . . .

  Instead:

  computation as interaction: inputs are 'awaited' or tracked and the outputs are actions that are dynamically raised while the process is executed (there is no *final result*)

# Interaction-based Paradigm

## Interactive Program

A collection of entities (agents, databases, network services, etc) that interact according to some interaction rules

- The interaction rules can be constrained by interfaces, protocols and quality of service (QoS) requirements (timeouts, confidentiality, etc)
- Instances of this model of interactive programming:
  - Event-driven programming
  - Reactive systems, Embedded systems
  - Client/server architectures
  - Software agents
- This model underlies distributed applications, user-interfaces design, web programming, and the incremental design of programs where parts of a program are refined during its execution.

# Interaction-based Paradigm

## Event-driven programming

- The control flow is determined by the occurrence of events

> Hardware events: mouse clicks, mouse mouvements, a key pressed, external signals coming from other devices, etc... Software events: messages issued from other programs or processes, etc.

- A typical architecture for an *event-driven* (or *event-based*) application consists of a main loop having two independent sections
  1. *event-detection*
  2. *event-handling*
- As for *embedded systems*, the first section is actually part of the *hardware* and is managed by means of interruptions

# Interaction-based Paradigm

## Event-driven programming

Event-driven programming is not bound to any specific programming paradigm.

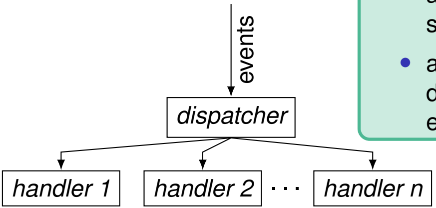
- Event-driven programs can be written in any high-level language, provided that the *event-driven* style is feasible.
- Object-orientation is not necessary
- Concurrency is not mandatory
- Requirements:
  - Catch signals, processor interruptions or, in GUI applications, mouse clicks
  - Managing an event queue to launch the appropriate event-handler

# Interaction-based Paradigm

## Event-driven programming

- Following the *event-handler* pattern is useful to implement this kind of applications.

### The *event-handler* pattern




- a *dispatcher* manages the sequence of events
- a collection of *handlers* dealing with the different events

# Interaction-based Paradigm

## Event-driven programming: example of *dispatcher*

main loop

exit

*handler* selection

```
do forever:   // the event loop
  get an event from the input stream

  if event.type == EndOfEventStream :
    quit   // break out of event loop

  if event.type == ...:
    call the appropriate handler, passing it
    event information as an argument

  elseif event.type == ...:
    call the appropriate handler, passing it
    event information as an argument

  else:   // unrecognized event type
    ignore the event, or raise an exception
```

# Interaction-based Paradigm

## Final remarks

Event-driven programming is massively used in GUI development. This is because most development environments provide assistants for implementing the event-driven pattern.

- Advantages:
  - It simplifies the programmers' burden by providing a default implementation for the main loop and management of the event queue.
- Disadvantages:
  - It promotes a too simple event-action model
  - It is difficult to extend
  - It is error prone: managing shared resources is difficult

# Emerging paradigms

- BIO-COMPUTATION: there are computational models inspired in biology
  - they use techniques that are inspired by biological systems as a basis for computation and programming
- QUANTUM COMPUTING: replace classical circuits by others that can take benefit from quantum effects (using quantum gates rather than logic gates)

# Matching PLs and Programming Paradigms

There are many multiparadigm PLs:

- **CoffeeScript** (2009): It is an OO functional and imperative language based on prototypes. CoffeeScript compiles into JavaScript.
- **Scala** (2003): Object-oriented, imperative and functional (used in Twitter together with Ruby).
- **Erlang** (1986): functional and concurrent (used by HP, Amazon, Ericsson, Facebook, . . . )
- **Python** (1989): functional (comprehension lists, lambda abstractions, fold, map) and object-oriented (multiple inheritance)

# Basic references

- Cortazar, Francisco. *Lenguajes de programación y procesadores*. Editorial Cera, 2012.
- Peña, Ricardo. *De Euclides a Java: historia de algoritmos y lenguajes de programación*, Editorial Nivola, 2006.
- Pratt, T.W.; Zelkowitz, M.V. *Programming Languages: design and implementation*, Prentice-Hall, 2001
- Scott, M.L. *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2008 (revised version).
- Schildt, Herbert. *Java. The Complete Reference*. Eight Edition. The McGraw-Hill eds. 2011

# References

## Implementation aspects


- “Programming Language Pragmatics”, M.L. Scott. (chapter 3)
- “Lenguajes de programación y procesadores”, Francisco Cortazar (chapter 1)
- “Programming Languages: design and implementation”, Pratt, T.W.; Zelkowitz, M.V. (chapters 9 and 10)