Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

Programming Languages and Paradigms

Francisco Ortín Soler

University of Oviedo

Content

- Programming Language, Compiler and Interpreter
- Features of Programming Languages
- Programming Paradigms
- Programming Technology
- The Language Selected

Programming Language

- A **programming language** is an artificial language for writing instructions, algorithms or functions that can be executed by a computer
- It is a language to **bring** the <u>human abstraction</u> level **close** to the <u>machine abstraction</u> level

Machine Language

1010010011010

Human Language

English
Spanish

Programming Language

```
class Person {
  …
}
```

Translator, Compiler and Interpreter

- A language **translator** is a program that translates source programs in a programming language into <u>equivalent</u> destination programs in another language
- A **compiler** is a language translator that translate <u>high-level</u> programs into low-level machine code
  - A compiler is a specific type of language translator
- An **interpreter** is a program that <u>executes</u> programs written in a specific programming language

Language Classification

- Programming languages are **classified** regarding their **features**
  1. Abstraction level
  2. Domain
  3. Concurrency support
  4. Implementation (compiled or interpreted)
  5. When type checking is performed
  6. How the source code is represented
- We will go into each classification in depth

Abstraction Level

1. **Abstraction level** (high / low)
   - Considers if the abstraction level is closer to the human abstraction lever (high) or to the computer abstraction level (low)
   - Assembly language and machine code are **low-level** abstraction languages
   - C is considered a **medium-level** abstraction language by some authors (others consider them low-level)
   - The rest of programming languages are commonly classified as **high-level** languages
   - Domain Specific Languages (DSLs) commonly provide an **even higher level of abstraction**

Domain

2. **Domain**: Languages can be general-purpose or domain-specific languages (DSLs)
   - **DSL**s are dedicated to a specific problem domain
   - Examples: SQL (databases), Logo (drawing), R (statistics) o Csound (music)
   - **General-purpose** languages are designed to build programs in any application domain
   - Examples: Java, C++, Python, C#, Pascal…
   - In the last few years, the use of DSLs have increased due to its use in Model Driven Development (MDD) and Domain Specific Modeling (DSM)

Concurrency Support

3. **Concurrency**: Languages that support the creation of programs as a <u>collection of interacting processes</u> or <u>threads</u>, which *may* be executed in parallel
   - Concurrent programs can be executed:
     - **In a single processor**, interleaving the execution steps of each computational process
     - **In parallel**, by assigning <u>each</u> <u>process</u> to <u>one</u> of a set of <u>processors</u>, which can be in the same machine or distributed across a computer network
   - <u>Concurrent languages</u>: Go, Ada, Erlang
   - Many languages do not provide concurrency as part of their syntax, but they do as part of their <u>standard</u> <u>library</u>: Java, C#, C++14, Python, Smalltalk
   - <u>Non concurrent</u>: C (the most used language in super-computing!) and C++ 2003

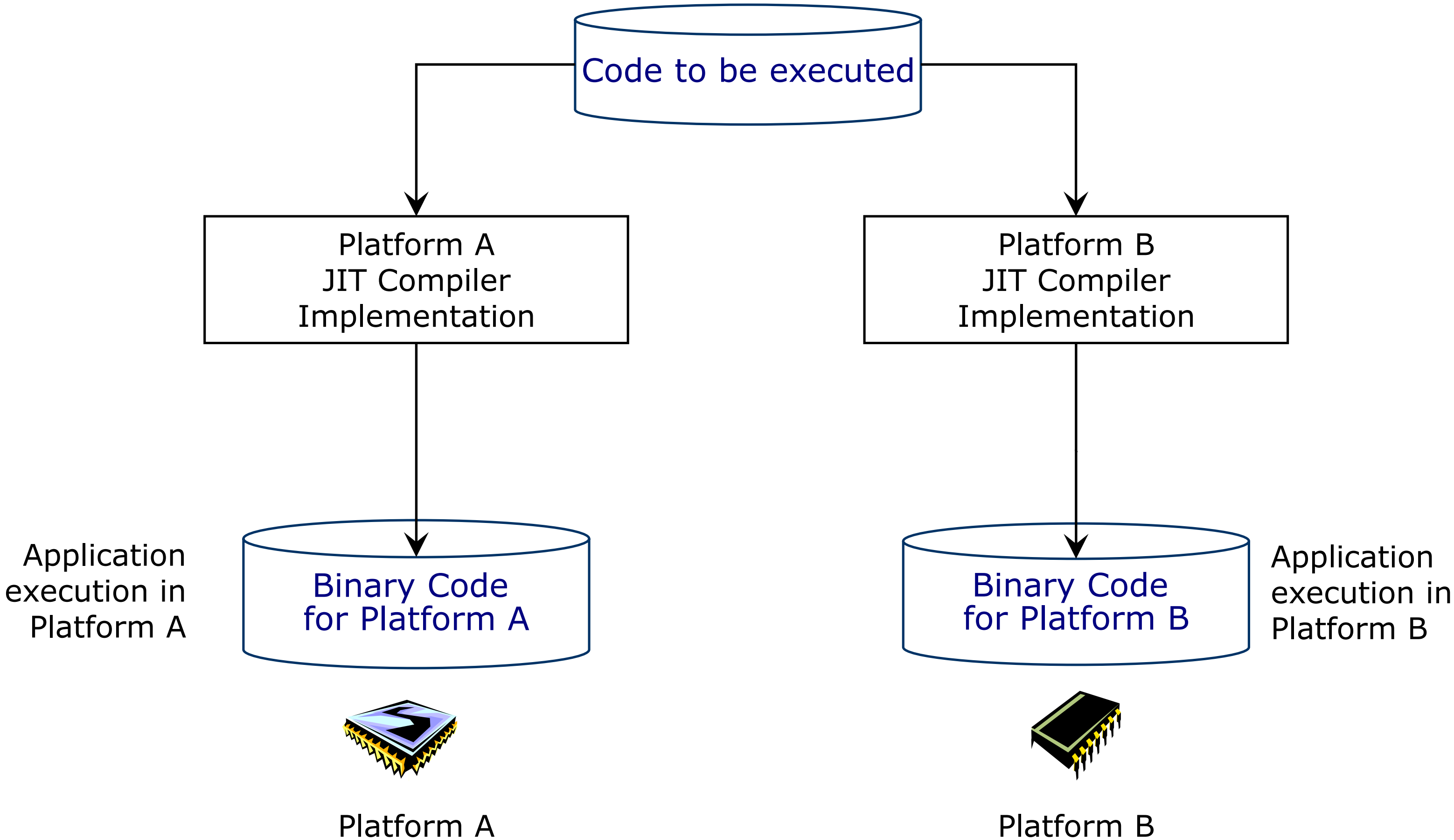
Implementation

4. Implementation: Compiled or Interpreted

This feature is **related to the language implementation**, rather than to the language itself

- The two alternatives are not mutually exclusive: Java and C# are compiled and "interpreted"
- **JIT** (Just In Time) **compilation** is a hybrid approach
  - A JIT compiler transform the code to be interpreted in specific-machine binary code, before its execution: Self, Visual Works Smalltalk, Java, .NET
  - When this transformation is performed prior to application execution it is then called **AoT** (Ahead of Time)

JIT Compilation



- NGen is a .NET utility to perform AoT compilation of .NET assemblies

Implementation

- Languages designed to be interpreted: Perl, Tcl, JavaScript, Matlab, PHP
- Languages designed to be compiled: C, C++, Go, Pascal, Eiffel, Ada
- Languages designed to be first compiled and then interpreted: Java, UCSD Pascal
- Languages designed to be JIT-compiled: Self, C#
- Languages with both compiled and interpreted implementations: Haskell, Lisp

Type Checking

5. **When type checking is performed**

   Type checking (checking the operations accepted by a variable) can be done **statically** (by the compiler) or **dynamically** (at runtime)

- The two alternatives are not mutually exclusive: Java is mostly statically typed, but also provides dynamic typing
- **Static** typing has the two major benefits
  - Earlier (compile time) type error detection
  - Better runtime performance (types do not need to be discovered and checked at runtime)
- **Dynamic** typing usually improves:
  - Runtime adaptability and flexibility
  - Dynamic meta-programming

Dynamic and Scripting Languages

- Dynamically typed programming languages are commonly named either:
  - **Scripting** languages: languages used to automate the execution of tasks, mainly consisting in gluing components in some host environment
    - (ba)sh, Awk, Perl, Tcl, ActionScript, Lingo, JavaScript…
  - **Dynamic Languages**: provide all the features to build full-fledged applications (database access, GUIs, native threads…) by only using that programming language
    - Smalltalk, CLOS, Python, Ruby, Dylan, Groovy, Lua
- Despite this difference, it is common to see both used as synonyms

Type Checking vs. Implementation

	Only static type checking	Both static and dynamic type checking	Only dynamic type checking
Compiled	C, Fortran, Pascal	C++ (RTTI), Ada, ObjectiveC, Scala	Python, Ruby 1.9+ (compiled and interpreted)
Interpreted		GHCi Haskell, OCaml (compiled and interpreted)	Smalltalk, Prolog, Ruby 1.8-

- Languages with <u>only static type checking</u> are <u>compiled</u>
- No fully-interpreted language provides only static type checking
- There may be <u>compiled languages</u> (without any interpretation) that provide <u>dynamic type checking</u>
- Languages that only provide <u>dynamic type checking are interpreted</u> (they could also be compiled before interpretation)

Source Code Representation

6. **How source code is represented**

   Languages could be visual or textual

- **Visual** languages represent their domain entities by means of a visual notation
- Examples are (Executable) UML, Lava, LabVIEW and VisSim
- There are many visual DSLs: Macromedia Authorware (multimedia), Game Maker (games), Scratch (teaching), Ladder Logic (electronic), MVPL (robotics)
- **Textual** languages use text files for their source code
- Examples are Java, C#, Haskell, Pascal…

Programming Paradigms

- We have already seen the most common programming language features used to classify them
- A programming **paradigm** is an **approach** to solve **programming** problems based on the **abstractions** and concepts used to represent programs (objects, functions, constraints, predicates…)
  - The programming paradigm is also used to classify programming languages
  - A programming language can support multiple paradigms (**multi-paradigm**)
  - A programming paradigm do **not** constitute a language feature

Imperative vs. Declarative

- **Imperative** and **Declarative**: Some authors consider each one a paradigm itself
- More appropriately, they represent a **paradigm classification**
  - Occasionally, even a language classification
- An **imperative** paradigm describes programs in terms of **statements** that change the **program** (machine) **state**
  - The programmer specifies **how** the computer must perform a task
  - Its abstractions are close to the machine abstractions
  - Examples of imperative languages: C, Java, C#, Pascal
  - Examples of imperative paradigms: Structured procedural-based (a.k.a., simply imperative) and object oriented

Imperative vs. Declarative

- **Declarative** paradigms describe programs by specifying **what** should be accomplished, <u>avoiding specifying **how** to do it</u>
  - Programmers **declare** what should be achieved
- Depending on the paradigm, different **abstractions** are used to specify the *what*
  - Functions, predicates, constraints, queries…
- <u>Examples</u> of **paradigms** usually identified as **declarative** are logic, functional and constraint programming
- <u>Examples</u> of **declarative languages** are Prolog, SQL, CLP(R), Maude, Haskell

Main Paradigms

- **Functional**: Declarative paradigm that abstracts programs as mathematical functions computing immutable data
- **Logic**: Declarative paradigm that abstracts programs by means of mathematical logic
- **Structural Procedural**-based: Imperative paradigm that provides three control flow structures (sequential, conditional and iterative), grouping code into procedures or subroutines
- **Object Oriented**: Commonly imperative paradigm that abstracts programs as objects (comprising data and methods) together with their interactions

Functional Paradigm

- Declarative paradigm that abstracts programs as mathematical **functions** computing <u>immutable data</u>
  - Data is never modified
  - A function returns new values without modifying the original ones, passed as a parameters
- A **program** is defined as a <u>set of functions invoking one another</u>
- Opposite to imperative programming, functions can **never** have **side effects**:
  - the value of an expression <u>only depends on the</u> value of its <u>parameters</u>
  - <u>always returning the same value</u> for the same values of the parameters (pure functional paradigm)
- **Recursion** is widely used instead of <u>iteration</u>

Functional Paradigm

- **Pure functional languages** avoid the use of (destructive) assignments and instruction sequencing
- Functional programming has its roots in **lambda calculus** defined by Church and Kleene in the 30s
- Actually, many commercial non-functional languages are including elements of this paradigm (e.g., C# 3+ and Java 1.8)
- Examples of pure functional languages are Miranda, Clean and Haskell
- Examples of functional languages are Scheme, Lisp, Erlang, ML (Standard ML, CamL, F#, OCaml)

Logic Paradigm

- Declarative paradigm that abstracts programs by means of **mathematical logic**
- The programmer describes knowledge by means of **logic rules** and **axioms** (facts)
- A theorem prover with **backward chaining** solves problems upon **queries**

```
mother(teresa, sandra).
mother(sandra, john).
mother(teresa, thomas).
father(thomas, sandra).
father(mike, thomas).
father(thomas, elisa).
```

Facts (axioms)

Rules

```
sibling(X, Y) :- father(Z, X), father(Z, Y), \=(X, Y).
sibling(X, Y) :- mother(Z, X), mother(Z, Y), \=(X, Y).
```

```
?- sibling(sandra, X).
X = elisa, X = thomas.
```

Query and solution

Logic Paradigm

- First described by John McCarthy (the creator of Lisp) in 1958
- The first language considered as logic was Absys developed in 1967
- The most used logic language is Prolog (70s), a language with many variants
  - **Concurrent**: Brain Aid Prolog (BAP), Muse, Aurora
  - **Constraint solving**: B-Prolog, CLP(R), Ciao Prolog, SICTus
- There are many **functional logic** languages that combine both paradigms such as ALF, Mercury, Mozart – Oz, Life and Toy

Structured Procedure-Based

- This paradigm is sometimes refer to as simply **imperative**
- It is the combination of **two historical paradigms**
  - Structured paradigm
  - Procedural paradigm
- The **structured** paradigm provides <u>three control flow structures</u>: sequential, conditional and iterative
  - <u>Jump</u> instructions are avoided
  - Improves program <u>readability</u> and <u>maintainability</u>
  - Control flow structures can be <u>nested</u>

Structured Procedure-Based

- The **procedural** paradigm defines the **procedure** or **subroutine** as the main mechanism of reutilization
  - A procedure is a <u>sorted sequence of statements</u>
- This paradigm **includes the existing structured paradigm**
  - This is the reason they are commonly identified as the same paradigm
- Includes the concept of variable **scope**, preventing erroneous variable accesses
- A procedure that returns a value is called **function**
- <u>Important</u>: It is <u>different from mathematical **functions**</u> in the functional paradigm
  - They can have <u>side effects</u>
  - <u>Higher order functions</u> are not commonly supported
  - <u>Closures</u> functions are not commonly supported

Structured Procedure-Based

- There are plenty of structured procedure-based programming languages
  - Algol, Ada, C, Pascal, Fortran, Cobol, PL/I
- Fortran, the first high-level general-purpose programming language, created by IBM in the 50s, follows this paradigm
  - The first versions only supported static memory (no stack)

Object Oriented Paradigm

- Paradigm that abstracts programs as <u>objects</u> (comprising data and methods) together with their <u>interactions</u>
- It is based on the idea of modeling **real objects** by means of **software objects**
  - The idea is to bring the <u>domain model</u> closer to the <u>program model</u>
- An object-oriented program is made up of a set of objects that interchange messages among them
- It is a commonly **imperative** paradigm
- Common **elements** of this paradigm are <u>encapsulation</u>, <u>inheritance</u>, <u>polymorphism</u> and <u>dynamic binding</u>

Object Oriented Paradigm

- There are two object models in this paradigm

1. **Class-based** model
   - Any object is an **instance** of a class
   - Classes are object **types**
   - Classes define both **structure** and **behavior** of objects
   - Examples: C++, Java, C#, Smalltalk, Eiffel
2. **Prototype-based** (object-based) model
   - The class concept is not provided; objects are the main abstraction
   - Object instantiation is obtained by **cloning a prototype** object
   - Examples: Self, Cecil, JavaScript, Python

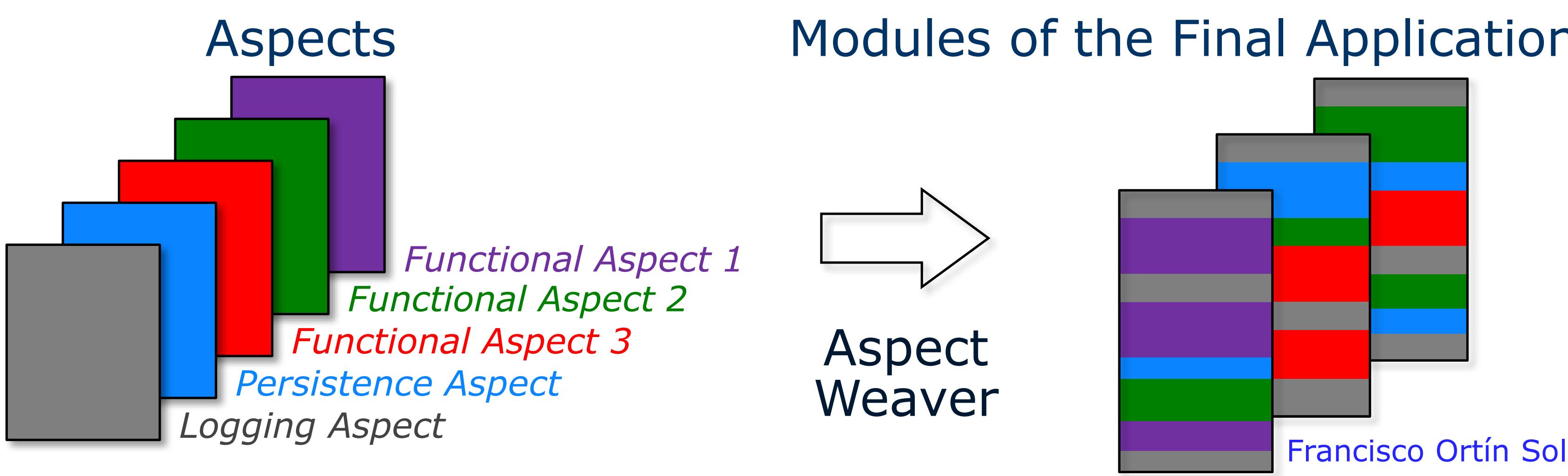
- An OO language is said to be **pure** when every abstraction in the language is an object: Smalltalk, Eiffel, Ruby

Other Programming Paradigms

- The are some other less-spread programming paradigms
  - Aspect Oriented
  - Constraint Solving
  - Real Time
  - Event Driven

Aspect Oriented

- Aspect-oriented programming is aimed at **modularizing** the **functionality** that is commonly **tangled** and **spread over** the application code (a.k.a. cross-cutting concerns)
  - Common examples of aspects are persistence, security, logging or tracing
- Most existing tools that support aspect orientation are based on code instrumentation techniques called **aspect weavers**



Aspect Oriented

- Mixes **imperative** program code with **declarative** aspect information
- The modularization improvement provided by aspect orientation provides **benefits** on software
  - Maintainability, readability, reusability (both aspects and components) and adaptability
- **AspectJ** is the most used aspect-oriented tool / language
- Aspect oriented programming has been included in production application servers such as JBoss and Spring
- **Aspect Oriented Software Development** (AOSD) is a software technology that suggests the use of aspect orientation in all the phases of software life cycle (http://www.aosd.net)

Constraint Solving

- Relations between variables are stated in the form of **constraints** ((in)equations)
  - Program execution returns (a set of) values as the result
- It is a **declarative** paradigm
- Constraints are expressed in different **domains**: Boolean, integer, rational, linear (vector spaces) and finite
- Languages are commonly implemented as
  - A **full-fledged** language: ECLiPSe, Mozart – Oz, OPL
  - As the **extension of another language**, usually logic: CLP(R), B-Prolog, Ciao Prolog, SICTus
  - As an **API** of an imperative language: Comet, Disolver, Gecode, ChocoSolver

Constraint Solving

- The following code is an example program in OPL (Optimization Programming Language) to solve the *(four) color map problem*

```
enum Country {Belgium, Denmark, France, Germany,
                  Netherlands, Luxembourg};
enum Colors {blue, red, yellow, gray};
var Colors color[Country];
solve {
     color[France] <> color[Belgium];
     color[France] <> color[Luxembourg];
     color[France] <> color[Germany];
     color[Luxembourg] <> color[Germany];
     color[Luxembourg] <> color[Belgium];
     color[Belgium] <> color[Netherlands];
     color[Belgium] <> color[Germany];
     color[Germany] <> color[Netherlands];
     color[Germany] <> color[Denmark];
};
```

Real Time

- A real-time system must perform **computations guaranteeing** response **within time constraints**, regardless of system load
- Real-time systems are commonly classified into
  - **Hard real-time**: Unfulfilling a time constraint means a total system failure
  - **Soft real-time**: Unfulfilling a time constraint degrades the quality of the system, but a recovery mechanism is provided continuing its execution (e.g., skip computing some data)
- Real-time programming languages provide abstractions for these time constraints: Ada and Real-Time Java
- It is common to use a native language combined with a specific operating system API: C + Real-Time Posix

Programming Technology

- The "*programming technology*" term is too broad
- In this course it is considered as

  *those **features** and **techniques** offered by different programming **languages** and **paradigms** to **design**, **build** and **maintain** applications in a **robust**, **secure**, **safe** and **efficient** way*

- We will study the foundations of the **object-oriented**, **functional** and **concurrent** paradigms
- For each **paradigm**,
  - The <u>features</u> and <u>techniques</u> offered to build applications appropriately are studied
  - We will analyze the current **trend** of commercial imperative languages <u>toward declarative paradigms</u>

Selected Language

- One approach is to use **different languages** in the course
  - It would be **difficult** to do so with 6 ECTS credits
- We have chosen one language
  - That provides multiple features and techniques of **different paradigms**
  - With <u>different implementations</u> in different platforms
  - **Standardized** so that third parties could implement the standard specification
  - **Widely used** nowadays, so that it involves a valuable asset in the student curriculum

C#

- **OO paradigm**: encapsulation, inheritance, polymorphism, generics, auto boxing
- **Functional paradigm**: lambda functions, higher-order functions, closures, a sort of continuations
- Standard API of **concurrent** programming and **parallelism**
- **Advanced features**: reflection, meta-programming, dynamic code generation, annotations or attributes, both static and dynamic typing, language integrated queries (lists comprehension)
- There exist implementations for Windows, Linux and Mac
- **ECMA** and **ISO** standard
- Wide use and rising trend since its release in 2002

C#

- Regarding the elements analyzed, C# is
  - High level
  - General purpose
  - Provides a standard concurrency and parallelism library (plus some elements in the language, e.g. `lock`)
  - First compiled, and then executed by a JIT-compiler virtual machine
  - (Mainly) statically and dynamically typed
  - Textual (not visual)
  - (Mainly) imperative
  - Its roots are object oriented, but it also includes elements of the functional paradigm

Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

Programming Languages and Paradigms

Francisco Ortín Soler

University of Oviedo