Advanced Programming - Appunti

Francesco Lorenzoni

September 2023

Contents

Chapter 1

Introduction

1.1 19 - Settembre

1.1.1 Info and Contact

Pagina del corso

1.1.2 Framwork

A software **framework** is a collection of common code providing generic functionality that can selectively overryidden or specialized by user code prvoiding specific functionality.

When using *framworks* htere is an **inversion of control**. Differently from what happens when using libraries, the program-flow is dictated by the framework, not the caller.

1.1.3 Design Patterns

A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is characterized by:

- **Name**
- **Problem Addressed**
- **Context** - Used to determine applicability
- **Forces** - Constraints or issues that the solution must address
- **Solution** - It must resolve all *forces*

1.2 20 - Settembre

Useful tool, to see preprocessor output, compiling, ecc.

1.2.1 Programming Languages

A **PL** is defined via **syntax**, **semantics** and **pragmatics**[1].

Syntax

Used by the compiler for *scanning* and *parsing*. The *lexical* grammar defines the syntax of token (e.g. "for" blocks, constants)

Semantics

Semantics might be described using natural language, which even if precise, allows amibguousity. Formal approches to semantics definition are:

1. Denotational - Mapping every syntactic entity with a mathematical entity
2. Operational - Defining a computation relation in a form $e \Rightarrow v$, where e is a program
3. Axiomatic - Based on Hoare-triples $Precondition \wedge Program \Rightarrow Postcondition$

However, they rarely scale to fully-fledged programming languages.

[1] the way in which the PL is intended to be used in practice

Pragmatics

Pragmatics include coding conventions, guidelines for elegant code, etc.

1.2.2 Programming Paradigms

Paradigms belong to languages *pragmatics*, not to the way the language is defined, i.e. not syntax nor semantics.

1. **Imperative**
2. **Object-oriented**
3. **Concurrent** - Processes, communication, ...
4. **Functional**
5. **Logic** - Assertions, relations, *strange sorceries*...

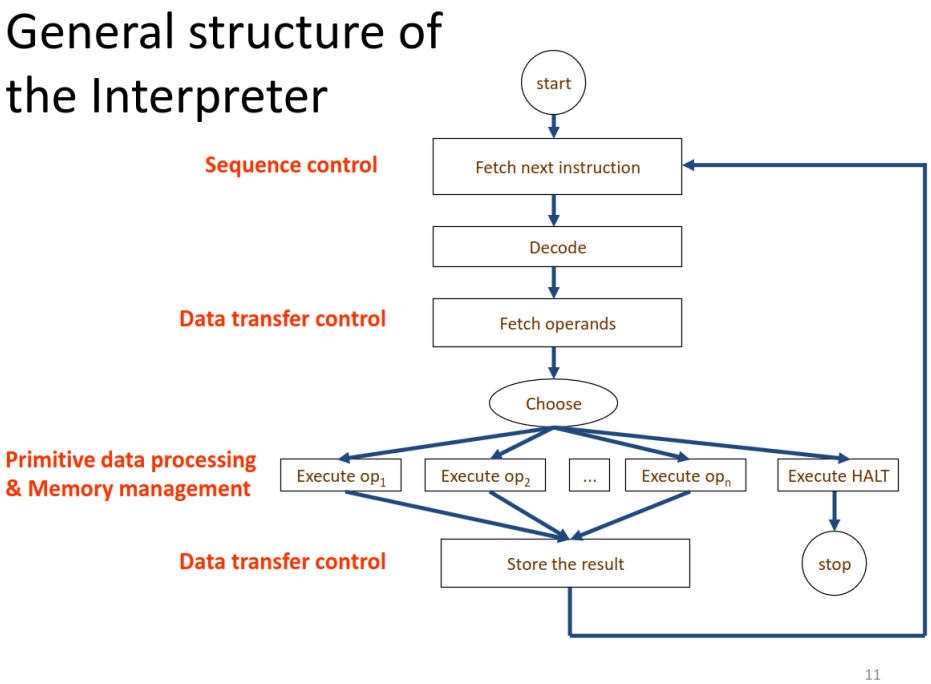
Modern PLs, provide constructs and solutions to program in all these paradigms

1.2.3 Implementing PLs

- Programs written in **L** must be *executable*
- Every language **L** implicitly defines and *Abstract Machine* M_L having **L** as a Machine Language
- Implementing M_L on an existing host machine M_O via compilation or interpretation (or both) makes programs written in **L** executables

An **Abstract Machine** M_L for L is a collection of data structures and algorithms which can perform the storage and execution of programs written in L.
Viceversa, M defines a language L_M including all programs which can be executed by the interpreter M.
There is a bidirectional correspondance between machines and languages components.

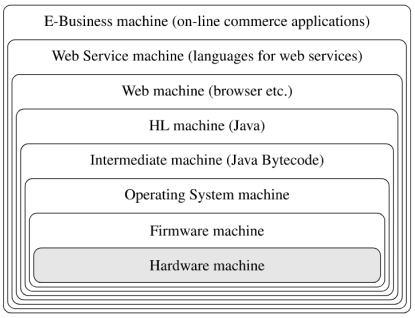
Primitive data processing	$\longleftrightarrow$	*Primitive data types*
Sequence control	$\longleftrightarrow$	*Control structures*
Data transfer control	$\longleftrightarrow$	*Parameter passing and value return*
Memory management	$\longleftrightarrow$	*Memory management*



In computer science one of the main focuses is **abstraction**, as can be seen in this hierarchical scheme.

Hierarchies of Abstract Machines

- Implementation of an AM with another can be iterated, leading to a hierarchy (onion skin model)
- Example:



A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Implementing PLs - Wrap Up

- L High-level programming language
- M_L Abstract machine for L
- M_O host machine

Pure Interpretation

...PIC HERE

M_L is interpreted over M_O. It isn't very efficient, mainly because of fetch-decode phases

Pure Compilation

...PIC HERE

L programs are translated into L_O, the machine laguange of M_O, hence, M_L is not realized at all and the programs are directly executed on M_O.
Compilation is more efficient than *Interpretation*, but produced code is larger

Both

...PIC HERE

All real languages use both *interpretation* and *compilation*,

Some languages, e.g. Java, use an intermediate Abstract Machine, called a *Virtual Machine*, which increases *Portability* and *Interoperability*.

Chapter 2

JVM

25 - Settembre

2.1 Runtime System

Every language defines and **execution model**, which is (partially) implemented by a **runtime system**, providing runtime **support** needed by both *compiled* and *interpreted* programs.

A **Runtime system** includes (eventually):

- Code:
  - in the executing program generated by the compiler
  - running in other threads/processes]
- Language libraries
- Operating system functionalities
- The interpreter/virtual machine itself

Runtime support can be needed for various reasons:

- Memory Management → Stack Management, Heap Management
- I/O operations → File System, Sockets, I/O devices
- Intercation with runtime environment
- Parallel execution (threads/processes)
- Dynamic binding type checking
- Dynamic loading and linking of modules
- Debugging
- ¿Code Generation?
- ¿Verification and Monitoring?

2.1.1 JRE

The **Java Runtime Environmnet** includes **JVM** and **JCL** (*Java Class Library*).

2.2 JVM

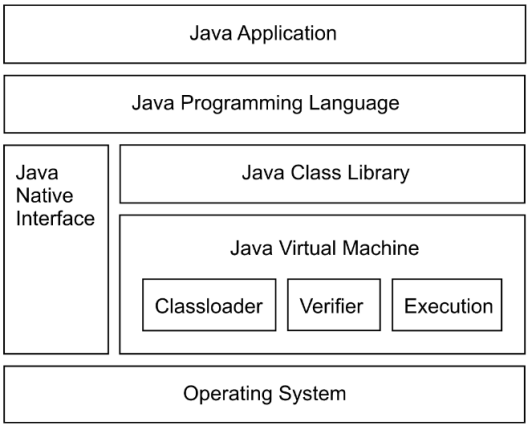
JVM is an *abstract* machine, defined by the documentation, which omits details on stuff like memory layout of runtime data area, garbage-collection, internal optimization, and even the representation of the `null` constant. The JVM specification, instead, defines precisely a machine indipendent *"class file format"* that all JVM implementations must support; it also imposes strong **synctatic** and **structural constraints** on the code in a class file.

The **JVM** is not *register-based*, instead it is a *multi-threaded* ***stack***[1] *based machine*. Id est the JVM pops intructions

[1] Not to be confused with the stack of activation records!

from the top of **operand stack** of the current frame, and pushes their result on the top of the **operand stack**. The **operand stack** is used to:

- Pass arguments to functions
- Return results from a function
- Store intermediate values while evaluating expressions
- Store local variables



2.2.1 Data types

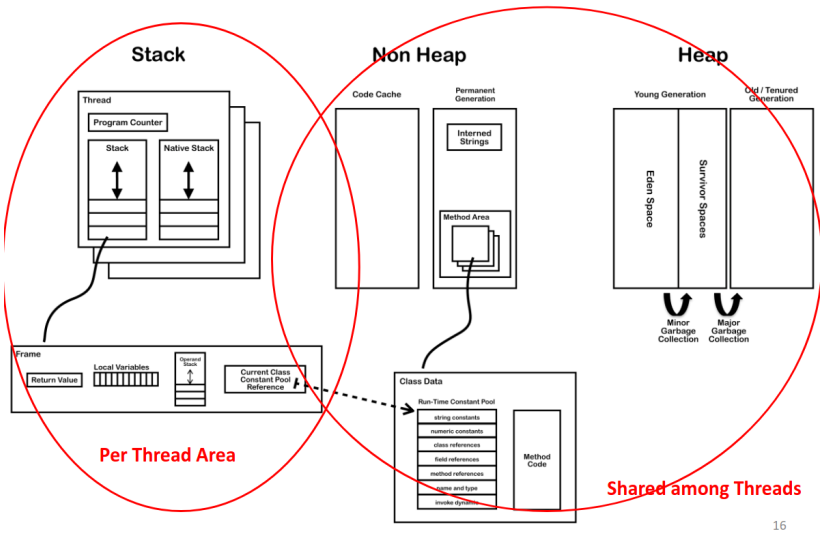
`.class` file are platform independent external represantions, which are represented internally by the JVM using simpler data types, which are implementation dependent.

- **Primitive types**
  - Numerica integral
  - Numeric floating point
  - boolean (support only for arrays)
  - internal (for exception handling)
- **Reference types**
  - Class types
  - Array types
  - Interface types

  No type information on local variables at runtime, there are only operand types specified by **opcodes** e.g. `iadd, fadd` , . . .
-

2.2.2 Threads

There are some runtime areas of the JVM related to a single thread, while others are shared among threads



All Java Programs are multithreaded, since there is at least a `main` thread running the user's program, and many

daemons:

- Garbage collector
- Signal Dispatching
- Compilation
- ¿ ... ?

JVM doesn't poses strong implementation constraints, by defining a precise abstract consistency model, including volatiles, allowing non-atomic longs and doubles, distinguishing working-memory and general store.

2.2.3 per-thread Data Areas

- **pc** pointer to next instruction in *method area*
  undefined if current method is native
- **Java stack**: stack of *frames* (or *activation records)*
- **Native stack**: used for invocation of natve functions through the *Java Native Interface* (*JNI*)

Considering the **structure** of **frames**, each one is composed by:

- **Local Variable Array** (32 bits) containing:
  1. Reference to `this`
  2. Method parameters
  3. Local variables
- **Operand stack**
- Reference to **Constant Pool** of current class

Differently from C/C++, where the **linking** phase is done before running an executable, java computes linking **dynamically** at **runtime**; this is achieved using **symbolic** references, which can be resolved using *static* (eager) or *late* (lazy) resolution.
Since the execution of a Java program must **not** depend on the JVM implementation, the JVM always behaves as if the implementation implies *lazy* resolution, even if the actual implementation provides static resolution instead.

2.2.4 shared data areas

Heap

- Memory for objects and arrays
- No explicit deallocation, it is demanded to the garbage collection.

Non-Heap

Memory for objects never deallocated

- Method area
- Interned strings
- Code cache for JIT

Just In Time (JIT) compilation refers to profiling as "hot" code areas of bytecode which may be executed many times, and storing the compiled native code in a cache in the *Non-heap* memory.

Method-area

Here `class` files are loaded. For each class a classloader reference and the following info from the `class` file are stored:

- Runtime Constant Pool
- Field data
- Method data
- Method code

Method area is *shared* among threads! Access to it mustbe *thread safe*.

This should a **permanent** area of the memory, but it may be **edited** when a new class is loaded or when a symbolic link is resolved by dynamic linking.

26 - Settembre

Constant Pool

Contains constants and symbolic references for dynamic binding. It is possible to see the constant pool of a compiled `.class` file using the command:

```
javap -v name.class
```

Displaying something resembling to:

```
   #1 = Methodref          #6.#14         // java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #17            // Hello World
   #4 = Methodref          #18.#19        // java/io/PrintStream.println:(Ljava/lang/
       String;)V
   #5 = Class              #20            // com/baeldung/jvm/ConstantPool
   #6 = Class              #21            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               sayHello
  #12 = Utf8               SourceFile
```

2.2.5 Loading

Loading is finding the binary representation of a class or interface type with a given name and creating a class or interface from it.
Class (or Interface) C creation is *triggered* by other classes **referencing** C or by methods (e.g. reflection). If not previously loaded, `loader.loadClass` is invoked.

There are 4 **Classloaders**:

1. **Bootstrap CL**: loads basic Java APIs
2. **Extension CL**: loads classes from standard Java extension APIs
3. **System CL**: loads application classes from *classpath*
   (default application CL)
4. **User Defined CLs**: can be used for:
   - runtime classes reloading
   - loading network, encrypted files or on-the-fly generated classes
   - supporting separation between different groups of loaded classes as required by web servers

Runtime Constant Pool

2.2.6 Linking

Linking includes *verification, preparation, resolution.*

1. **Verification** multiple checks at runtime, e.g. operand stack under/overflows, validity of variable uses and stores, validity arguments type. Details later on
2. **Preparation** Allocation of storage
3. **Resolution**[2] resolve symbol references by loading referred classes/interfaces

Verification is a relevant part of JVM Specification, it is described in 170pp over a total of 600pp. When a class file is loaded there is a *first* verification pass to check formatting, there is a *second* one when a class file is linked regarding only not instruction-dependant checks. During the linking phase there is a data-flow analysis on each method (*third check*), and lastly (*fourth check*) when a method is invoked for the first time.

2.2.7 Initialization

`<clinit>` initialization method is invoked on classes and interfaces to initialize class variables; it also executes static initializers. `<init>` initialization method instead is used for instances.

[2] Optional, it may be postponed till first use by an instruction

Chapter 3

JVM Instr Set & JIT

3.1 Instruction Set

26 - Settembre

Let's consider the instructions **format**. Each instr may have different *"forms"* supporting different kinds of operands. For example there are different forms of `iload` (i.e. push).

Runtime memory contains - Local variable array (frame) – Operand stack (frame) – Object fields (heap) – Static fields (method area)

Note that Java instructions are explicitly typed through **opCodes**, e.g. `dload`,`iload`,`fload`.
opCodes are bytes, allowing only for 256 distinct ones; hence it is impossible to have for each instruction on opCode per type. The JVM specification indicates a selection of which types to support for each op instruction, and not supported types have to be converted; resulting in the Instruction Set Architecture to present non-orthogonality.
Types like `byte`, `char` and `short` are usually converted to `int` when performing computations.

27 - Settembre

3.1.1 Invoking methods

`invokevirtual` causes the allocation of a new frame, pops the arguments from the stack into the local variables of the caller (putting this in 0), and passes the control to it by changing the `pc`.

- A resolution of the symbolic link is performed
- `ireturn` pushes the top of the current stack to the stack of the caller, and passes the control to it. Similarly for `dreturn`, `freturn` ...
- `return` just passes the control to the caller

There are 4 others kinds of method invocation:

- `invokestatic`: call methods with `static` modifier; *this* is not passed
- `invokespecial`: call constructors, `private` methods or *superclass* methods; *this* is always passed
- `invokeinterface`: identical to `invokevirtual`, but used when the method is declared in an interface, thus a different lookup is required
- `invokedynamic`: introduced in Java 7 to support dynamic typing[1]

(...)

3.2 JIT

AOT **Ahead of Time** *Compilation* leads to better performance in general, exploiting hardware features and variables allocation without runtime lookup; While **Interpretation** facilitates interactive debugging and testing: it allows

[1] lambda functions related?

command-line invocation.

JIT aims to get the advantages of both.

JIT differs from *AOT* since it runs in the same process of the application and competes with the app for resources, thus compilation time for JIT is more relevant than for an AOT Compiler. Besides, a JIT compiler doesn't verify classes at compile time, it is a task performed by the JVM at load time. JIT can exploit new optimization possibilities, e.g. *deoptimization* and *speculation*. A JIT takes bytecode as input and outputs machine code that the CPU executes directly.
Wrapping up:

- Code starts executing interpreted with no delay
- Methods that are found commonly executed (*hot*) are JIT compiled
- Once compiled code is available, the execution switches to it.

To identify *hot* methods, there is a **threshold** on two *per-method* counters:

1. Times the method is invoked
2. Times a brach back to the start of a loop is taken in the method

A tradeoff between "fast-to-start-but-slow-to-execute" interpreter vs "slow-to- start-but-fast-to-execute" compiled code is managed by a multi tier system.

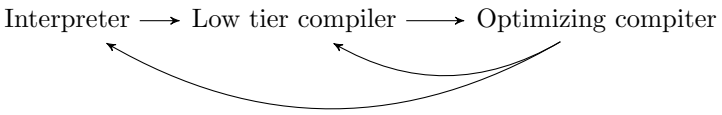
(...)

3.2.1 Deoptimization and Speculation

Usually method executions pass in three phases:

Interpreter $\longrightarrow$ Low tier compiler $\longleftrightarrow$ Optimizing compiter

But sometimes **deoptimization** can happen, i.e. :



Chapter 4

Component-Based software

2 - Ottobre

Component software indicates **composite systems** made of **software components**. In short, component software allows reuse, improving reliability[1] and reducing costs.
Bertrand Meyer suggests some guidelines regarding Object-Oriented software construction (1997):

1. modular
2. reliable
3. efficient
4. portable
5. timely

4.1 Definitions

*A **software component** is a unit of composition with contractually specified **interfaces** and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

A **contract**, A specification attached to an **interface** (component specification) that mutually binds the clients and providers of the components.

Context dependencies are specification of the deployment environment and run-time environment. This goes beyond the simple interfaces required and provided which are specified in the contract. Context dependencies include required tools, platform and resources.

Deployed independently means that a component can be plugged or unplugged from an architecture, even at runtime in some cases. It is common-practice to deploy "small components" called connectors, to resolve situations where two components supposed to interact do not provide identical interfaces, creating the need for a intermediary.

composition by third party means that a component may interact with third parties components without knowing the internals of such components.

4.2 Concepts of Component Model

- **Component interface** describes the operations implemented and exposed by the component
- **Composition mechanism** How components can be composed to work together to accomplish a task
- **Component platform** A platfom for the development of the compnents

Concepts should be laguage/paradigm agnostic, laying the ground for language interoperability.

The ancestors of Components are **Modules**, whose support has been introduced in Java 9, but isn't very common. Some concepts related to modules can be found in more modern notions such as classes, components and packages. For example, objects inside a module are visible to each other, but not visible from outside unless exported. Modules

[1] Industries may even require to use *certified* components

worked — pretty much like classes — as abstraction mechanism → *collection of data with operations defined on them.* In OOP the concept of **inheritance**, unknown in modules, is introduced.

4.3 Components and Programming Concepts

Components can be anything and can contain anything, they can be *collections of* classes, objects, functions/algorithms, data structures.

¡Note that ***OOP*** $\neq$ ***COP!***

OOP isn't focused on **reuse**, instead its focus in onto appropriate domain and problem representation.

- **Component Specification** describes the behavior (as a set of *Interfaces*) of a set of Component Objects and defines a unit of implementation.
- **Component Implementation** is a realization of *Component Specification* which can be independently deployed[2].
- **Installed Component** is an installed (i.e. *deployed*) copy of a *Component Implementation.*
  A Component Implementation is deployed by registering it with the runtime environment
- **Component Object** is an instance of a *Component Implementation.* It is a runtime concept, an object with its own data and unique identity. Ideally, it is the "thing that performs the implemented behaviour". An *Installed Component* may have multiple *Component Objects*

Some examples of successful components are Plugin architectures, Microsoft's Visual Basic, Operating Systems, Java Beans, and others. It is clear that components can be purchased by independent providers and deployed by the clients, and that multiple components can coexist in the same installation. Besides, components exist on a level on abstraction where they directly mean something to the deploying client.

Recalling the comparison with modules, while modules are usually seen as part of a program, *components are parts of a system.*

[2]It does **not** mean that it cannot have *dependencies* nor that it must be a *single file*

Chapter 5

Java Beans

5.1 3 - Ottobre

"A **Java Bean** is a *reusable* software component that can be manipulated visually in a builder tool."

Typically a Bean has a GUI representation but is not necessary. What is necessary instead for a class to be recognized as a bean is that it:

- has a public constructor with no arguments
- implements `java.io.Serializable`
- is in a `jar` file with a *manifest* file that contains: `Java-Bean: True`

Beans can be **assembled** to build a new bean or application, writing clue code to wire beans together. *Connection-oriented* programming is based on the **Observer** or (*Publish-Subscribe*) paradigm. *Observers* come into play when there is a $1 : N$ dependency between objects and one of them changes state, creating the need for the others to be notified and updated. Beans must be able to run in a *design environment* allowing the user to customize aspect and behaviour. Beans provide support for some standard features:

1. **Properties** e.g. color. **Bounded** properties generate an *event* of type `PropertyChangeEvent`, while **constrained** can only change value if none of the registered *observers* "poses a veto", by raising an *exception* when they receive the `PropertyChangeEvent` object.
2. **Events**: The **Observer** pattern is based on *Events* and *Events listeners*. An *event* is an object created by an *event source* and propagated to the registered *event listeners*. Sometimes event **adaptors** can be placed between source and listener, which might implement queuing mechanism, filter events, demuxing from many sources to a single listener.
   - **Design Patterns for Events**

     ```
     public void add<EventListType>(<EventListType> a)
     public void remove<EventListType>(<EventListType> a)
     ```
3. **Customization**
4. **Persistence**
5. **Introspection**: process of analyzing a bean to determine capabilities. There are implicit methods based on *reflection*, *naming conventions* and *design patterns*, but can be simplified by explicitly defining info for the builder tool in the `<BeanName>BeanInfo` class. Such class allows exposition of features, specifying customizer class, segregate feats in normal/expert mode, and some other stuff.
   - **Design Patterns for Simple Methods**

     ```
     public <PropertyType> get<PropertyName>();
     public void set<PropertyName>(<PropertyType> a);
     ```
   - **Design Patterns for Simple Methods**

     ```
     public java.awt.Color getSpectrum (int index);
     public java.awt.Color[] getSpectrum ();
     public void setSpectrum (int index, java.awt.Color color);
     public void setSpectrum (java.awt.Color[] colors);
     ```

Chapter 6

Reflection

9 - Ottobre

6.1 Introduction and Definitions

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. Another dimension of reflection is if a program is allowed to **read only**, or also to **change** itself.

- **Introspection** is the ability of a program to observe and therefore reason about its own state
- **Intercession** is the ability for a program to modify its own execution state or alter its own interpretation or meaning
- **Reification** is the mechanism of encoding execution state into data, which is needed by both *introspection* and *intercession*

.

Structural reflection is concerned with the ability of the **language** to provide a complete *reification* of both the *program* executed and its *abstract data types*.
Behavioral reflection is concerned instead with the reification of its[1] *semantics* & *implementation* (processor) and the data and implementation of the *run-time system*.

6.2 Uses and drawbacks

6.2.1 Uses

- *Class Browsers* need to be able to enumerate the number of classes
- *Visual Development Environments* can exploit type info available in reflection to aid the developer in writing correct code
- *Debuggers* need to be able to examine private members on classes
- *Test Tools* exploit reflection to ensure a high level of code coverage in a test suite
- *Extensibility Features* an app may make use of external, user-defined classes by creating instances of extensibility objects.

6.2.2 Drawbacks

- **Performance Overhead**
- **Security Restrictions**
- **Exposure of internals**

6.3 Reflection in Java

Java supports **introspection** and **reflexive invocation**, but not *code modification*.

[1] referred to a **language**

6.3.1 Introspection

The JVM mantains for every type an associated object of type `java.lang.Class` which "*reflects*" the type it represents, acting as entry point for reflection, since it provides all info needed:

- Class name and modifiers
- Extended superclasses and implemented inferfaces
- Methods, fields, constructors, etc.

To retrieve such `java.lang.Class` object it is sufficient to do `Object.getClass()`. `Class` objects are constructed automatically by the JVM as classes are loaded.

Using `java.util.reflect.*` it is possible also to retrieve class **Members** i.e. *fields, constructors* and *methods*. The extensive `java.util.reflect.*` API provides many *methods* to achieve this which will not be reported here.
There is a class for each Member

- `java.util.reflect.Field`: access type info and set/get values.
- `java.util.reflect.Method`: type info for parameters and return type; invoking method on a given object.
- `java.util.reflect.Constructor`: note that constructors have no return values and invocation creates a new instance of the given class.

6.3.2 Program Manipulation

By now we have talked only about **introspection** in java, but reflection can be used also to create objects of a type not known at compile time, or to access members (access fields or invoke methods) unknown at compile time.

Chapter 7

Annotations

9 - Ottobre

In java, `static`,`private`,... modifiers are *meta-data* describing properties of program elements. **Annotations** can be understood as (user-) definable modifiers. They are composed by one or two parts:

1. **name**
2. finite number of **attributes** i.e. `name=value`. There may be no attributes.

The syntax is the following:

```
    @annName             // e.g. Override
    @annName{constExp}   // shorthand for @annName{value=constExp}
    @annName{name_1 = constExp_1, ..., name_k = constExp_k}
```

`constExp` are expression which can be evaluated at *compile time*. Besides, attributes have a *type*, thus the supplied values have to convertible to that type.
Annotations can be applied to almost any syntactic element, from packages to parameters and any type use.

7.1 Defining annotations

```
@interface InfoCode {
    String author ();
    String date ();
    int ver () default 1;
    int rev () default 0;
    String [] changes () default {};
}
```

This defines the custom annotation InfoCode, imposing some fields possibly with default values. It can be used as follows:

```
@InfoCode(author="Beppe", date="10/12/07")
    public class C {
    public static void m1() { /* ... */ }
    @InfoCode(author="Gianni",
        date="4/8/08", ver=1, rev=2)
    public static void m2() { /* ... */ }
}
```

Chapter 8

Polymorphism

Polymorphism basically means *"many forms"*, where *forms* are **types**. Thus there may be *polymorphic* function names, or *polymorphic* types.

There are many "flavors" of polymorphism, many variations. Two main kinds opposed to each other are *ad hoc* and *universal* polymorphism, which however, may coexist:

- **ad hoc** PM indicates that a single function name denotes different algorithms, determined by the actual types.
- **universal** PM indicates a single algorithm (solution) applicable to objects of different types.

When PM is taken into account, it is crucial to consider when happens the **binding** between a function name and the actual code to be executed:

- compile time; *static/early binding*
- linking time
- execution time; *late/dynamic binding*

In general the earlier the binding happens, the better (for debugging reasons). If the binding spans over more phases (e.g. *overriding* in Java), as a convention we consider the **binding time** the last phase.

8.1 Classification

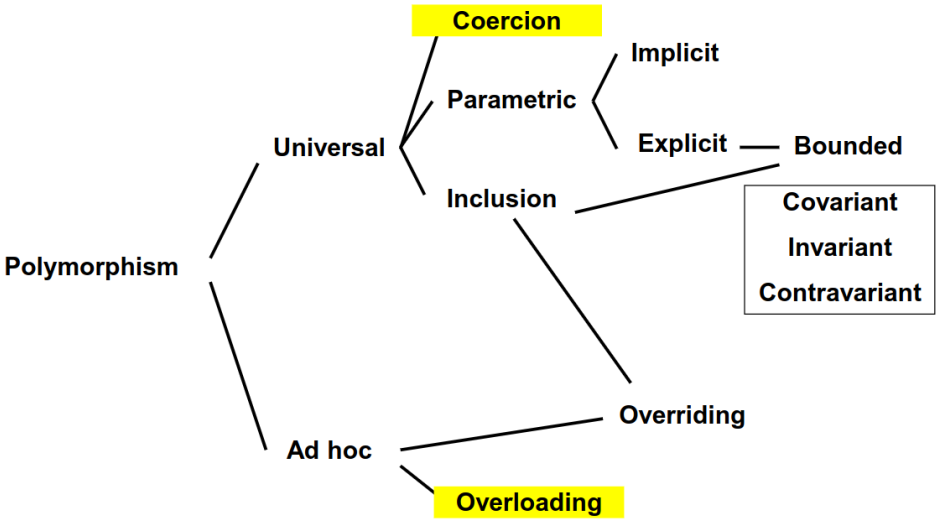


Figure 8.1: Polymorphism classification

8.1.1 Overloading

Overloading is present in every language for basic operators $+ - *$..., and sometimes is supported for user-defined functions, and in some languages it is even allowed the overloading of primitive operator by user-defined functions.

Since this falls under the **ad hoc** polymorphism family, the code to be executd is determined by the type of the arguments; the binding can either happen at *compile* or at *runtime*, depending on the typing of the language, whether it is static or dynamic.

```
// C language doesn't allow overloading for user-defined functions
int sqrInt(int x) { return x * x; }
double sqrDouble(double x) { return x * x; }

// Overloading in Java & C++
int sqr(int x) { return x * x; }
double sqr(double x) { return x * x; }
```

Haskell introduces **type classes** for handling overloading in presence of type inference

8.2 Coercion

Coercion is the automatic (implicit) conversion of an object to a different type, opposed to casting which is explicit instead. Coercion allows a code snippet to be applied of arguments of different (convertible) types. Sometimes coercion is allowed only if there is no **information loss**.

```
double sqrt(double x){...}
double d = sqrt(5) // applied to int
```

´

8.3 Inclusion Polymorphism

Inclusion polymorphism is also known as *subtyping polymorphism* or ***inheritance***. It is ensured by *Barbara Liskov*'s **substitution principle**:

> *A subtype object can be used in any context where a supertype object is expected*

Methods and fields defined in a superclass may be invoked and accessed by subclasses if not redefined (see *Overriding*).

8.4 Overriding

In Java a method m of a class A can be redefined in a subclass B of A.

Overriding introduces ad hoc polymorphism in the universal polymorphism of inheritance. Notice that overriding requires the final binding to happen at runtime: it happens through the lookup done by `invokevirtual` in the JVM.

8.5 C++ v Java

```
class A {
 public:
  virtual void onFoo() {}
  virtual void onFoo(int i) {}
};
class B : public A {
 public:
  virtual void onFoo(int i) {}
};
class C : public B {};
int main() {
  C* c = new C();
  c->onFoo();
  // Compile error - doesn't exist
}
```

The equivalent code in Java compiles, because in java invokes the function `onFoo()` with no arguments defined in the superclass A. In C++ instead, the function onFoo(int i) defined in B is found and stops the search, but there is arguments type mismatch, thus it doesn't compile. This happens because in C++ the method lookup is based on the method *name*, not on its *signature*.

8.6 C++ Templates

They are similar to *Generics* in Java, they are used as function and class templates each concrete instantiation produces a copy of the generic code, specialized for that type: monomorphization. In java Generics, instead, **type erasure** happens at runtime, i.e. type variables `T` are replaced by `Object` variables.
Templates support parametric polymorphism and type parameters can also be primitive types (unlike Java generics)

```
    template <class T> // or <typename T>
    T sqr(T x) { return x * x; }
```

Assuming to invoke `sqr(T x)` on variables of different types, the compiler will generated a specific code for each type used. This works even on user-defined types; check the following code for an example:

```
class Complex {
 public:
  double real;
  double imag;
  Complex(double r, double im) : real(r), imag(im){};
  Complex operator*(Complex y) {  // overloading of *
    return Complex(real * y.real - imag * y.imag,
                   real * y.imag + imag * y.real);
  }
};

Complex cc = sqr(c); // legal and produces a function "Complex sqr(Complex x) {...}"
```

It is important to check for type ambiguosity; in the following example, it is highlighted a case where it's not clear whether it is `i` to be converted to `long` or `m` to `i`.

```
template <class T>
T GetMax(T a, T b) {  return (a > b) ? a : b; }
...
n = GetMax(l, m);        // ok: GetMax<long>
// v = GetMax(i, m);     // no: ambiguous
v = GetMax<int>(i, m);   // ok
```

8.6.1 Macros

Macros can be exploited to achieve *polymorphism* and can have the same effect of the templates, but notice that macros are executed by the preprocessor[1] and are only **textual substition**, there is no parsing, no static analysis checks or whatsoever.

```
#define sqr(x) ((x) * (x))
int a = 2;
int aa = sqr(a++); // int aa = ((a++) * (a++));
// value of aa? aa contains 6  :(

#define fact(n) (n == 0) ? 1 : fact(n-1) * n
// compilation fails because fact is not defined
```

16 - Ottobre

8.6.2 Specialization

A template can be **specialized** by defining a template with the same name but with more specific parameters (*partial specialization*) or with no parameters (*full specialization*). This is kinda similar to *Overriding*, leaving to the compiler the choice of the most appropriate template.

```
/* Primary template */
template <typename T> class Set {
        // Use a binary tree
    };
/* Full specialization */
template <> class Set<char> {
        // Use a bit vector
```

[1] Macro expansion can be seen using the option *-E* when compiling

```
    };
/* Partial specialization */
template <typename T> class Set<T*> {
        // Use a hash table
    };
```

Templates can be used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.

Template compilation happens *on demand*: the code of a template is not compiled until an instantiation is required, however in case of *fully-specialized* template, the compiler treats the template as a function, thus it generates its code **regardless** whether it is ever used or not.

Note that in C/C++ while method *prototypes* usually are in a separate `.h` file, the compiler needs the template *declaration* and *definition* in the same place to instatiate it.