Advanced Programming

Federico Bruzzone

21 ottobre 2022

Indice

1 Informazioni generali

Scopo del corso

- Scoprire il concetto di separazione dei compiti;
- Imparare a programmare decomponendo le funzionalità del SW;
- Imparare ad ottimizzare il SW separandone le funzionalità;

Materiale di riferimento

- i licidi del corso;
- Ira R. Forman and Note B. Forman. Java Reflection in Action Manning Publications, October 2004;
- Ramnivas Laddad. AspectJ in Action: Pratical Aspect-Oriented Programming. Manning Pubblications Company, 2003;

2 Computational Reflection

2.1 Computational Reflection

2.1.1 A first definition

Computational reflection can be intuitively defined as:

"The activity done by a SW system to represent and manipulate its own structure and behavior"

The reflective activity is done analogously to the usual system activity

2.2 Reflection

2.2.1 Historical Overview

In the sisties

- Research field: artificial intelligence;
- First approaches to relection: intelligent behavior;

In the eighties

- Research filed: programming languages;

- Brian C. Smith, he introduces the reflection in Lisp (1982 and 1984), the reflective tower has been defined;
- Several reflective list-oriented languages have been defined (they exploit the quoting machanism);

In the meanwhile

- Research field: logic programming;
- the meta-programming takes place in PROLOG;

Between the eighties and the nineties

- Research fild: object-oriented programming languages;
- Pattie maes defines the computational reflection in OOPL (1987);
- Several people move from Lisp to OO:
  - P. Coite, ObjVLips (1987)
  - A. Yonezawa, ABCL-R (1988)
  - J. des Rivières e G. Kiczales MOP for CLOS (1991)
- SmallTalk is elected as the best reflective programming language

In te nineties

- Research field: typed and/or compiled object-oriented programming languages;
- Shigeru Chiba realizes OpenC++ (1993-1995), OpenJava (1999);

In the 1997

- Gregor Kiczales et al. defined the aspect-oriented programming and the story ends;

2.3 Computational Reflection

2.3.1 Reflection à la Pattie Maes

Pattie Maes has pioneered the filed

- a **computational system** is a system that can reason about and act on its applicative somain;

- a computational system is **causally connected** to its domain if and only if a change to its domain is reflected on it and vice versa;
- a **meta-system** is a computational system whose applicative domain in another computational system;
- **reflection** is the property of reasoning about and acting on itself;

therefore

- a **reflective system** is a meta system causally connected to itself;

2.3.2 Reflective system

From the definition, we can evince that a reflective system is:

- a software system logically layered into two or more levels respectively called base-level and meta-levels;
- the system running in a meta-level observes and manipulates the system running in the underlying level (reflective tower);

Characteristics

- the system running in the base-level is unaware of the existence and of the work of the systems running in the overlying levels;
- a meta-level system acts on a representation (called the system running in the underlying levels; and
- a system and its reification are causally connected and therefore, they are kept mutually consistent

2.3.3 Reflective system: Base- and Meta-levels

A meta-level system refies what it is implicit (e.g. mechanisms and structure) of the underlying base- or meta-level

2.3.4 How to Characterize a Reflective System

The reflective systems can be classified based on:

- what and when

What kind of reflective actions the system can carry out:

- structural and behavioral reflection;

- introspection (just to observe) and intercession (to alter)

When the meta-level entities exist:

- compile-time
- load-time; and
- run-time

2.3.5 Behavioral and structural reflection

The behavioral reflection allows the program of monitoring and manipulating its own computation, e.g.:

- to trap a method call and activating a different method instead;
- to monitor the object state;
- to create new objects, and so on

These activities can take place at run-time without a specific support

The structural reflection allows the program of inspecting and altering its own structure, e.g.:

- the code of a method can be modified or removed from the class;
- new methods and field can be added to a class, and so on;

These activities need a specific support by the execution environment (from the VM, RTE, ...) to be carried out at run-time

2.3.6 Reification

The base-level entities (referents) are reified into the metalevel, i.e., they have a representative into the meta-level

Such a representative, called reification, has to:

- support all the operations and have the same characteristics of the corresponding referent;
- be kept consistent to its referent (causal connection);
- be subjected to the manipulations of the meta-level entities to protect the base-level entities from potential inconsistency

Any change carried out on the reification has to be reflected on the corresponding referent.

2.4 To Develop a Reflective System

Jacques Ferber [2] has raised some issues that the developers must take in consideration:

- which kind of entities should be reified?
- what and how it is implemented the causal connection?
- when does the execution shift to the meta-level?

2.5 Which Kind of Entities Should Be Reified?

It depends on the programming language:

- functional: lambda expression/closures, environment, continuations, and so on ...;
- object-oriented: objects, methods, classes, messages and so on ...;
- concurrent and object-oriented: threads, processes, schedulers, monitors, and so on ...;
- distribution: namespaces, proxies, mailers, and so on ...

2.6 What and How It Is Implemented the Causal Connection?

It depends on when the reflective activities take place:

- atrun-time: the causal connection is explicit and must be maintained by an entities super-parties, e.g., by the virtual machine or by the run-time environment;
- at compile-time: the causal connection is implicit, base-level and meta-levels are merged together during a preprocessing phase;
- at load-time: in this case the causal connection behaves as in the case, reflection takes place at compile-time;

Most of the times, the supported reflective activity is related to observe (introspection) the base-level system so the causal connection become unilateral and can be managed by the metaentities.

2.7 When Does the Execution Shift to the Meta-Level?

Switching among levels depends on:

- which entities are reified;
- when such entities are reified; and
- how the causal connection is managed

The shift-up and-down actions

- the shift-up and-down actions.

When

- an observed element changes; or
- an action is going to be done;

the computational flow passes into the meta-level (shift-up)

Instead

- the computational flow goes back (shift-down) on the meta-level program decision

Usually, the shift-up action is managed by call-backs

3 Reflection in OO Programming Languages

3.1 Structural and Behavioral Reflection

Structural Reflection

- Object creation and init
  - constructor
  - prototype
  - meta-classes
- Class manipulation
  - to add or remove fields
  - to add or remove methods
  - to change the super class

Behavioral Reflection

- message sending
  - classes and inheritance
  - prototypes and delegation
  - errors
  - encapsulations
  - proxies
  - meta-objects

3.2 Structural Reflection

The objects running in the meta-level, called **meta-objects** are associated to all (or just to some of) the objects running in the base-level, called **referents**.

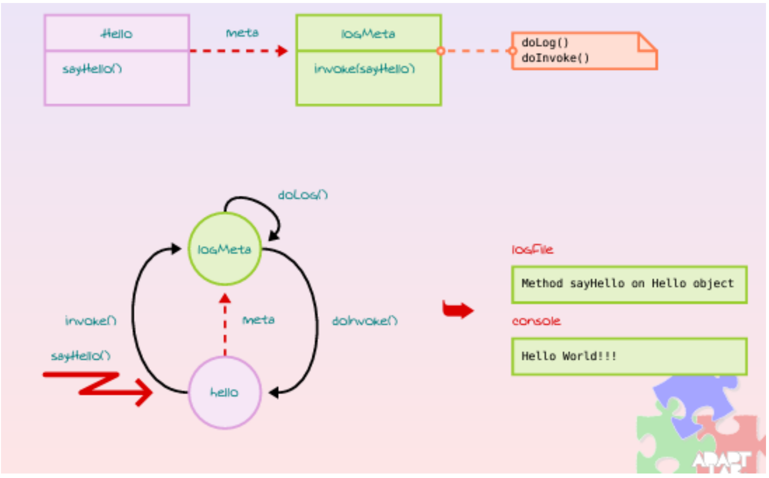
The connection among referents and meta-objects is called **causal connection** when it is a two-way link or **meta-connection** when it is a one-way link.

The meta-objects exist at run-time and extend or modify the semantics of some mechanisms:

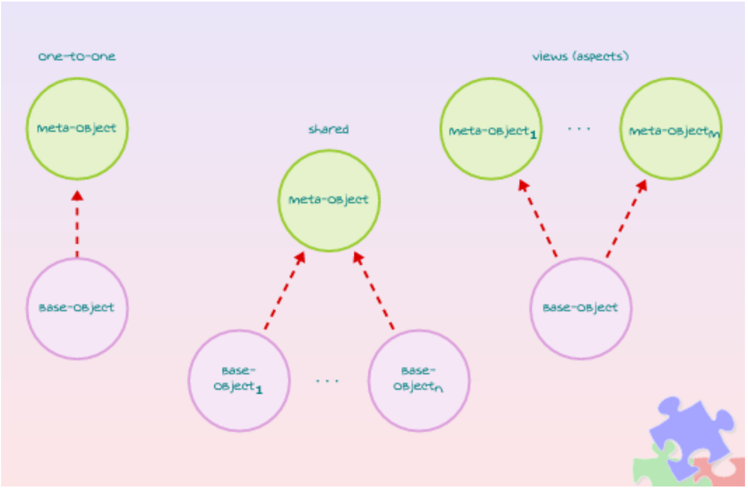
- method invocation, field access, object creation, and so on

The **MOP** is the set of messages that a meta-object can understand

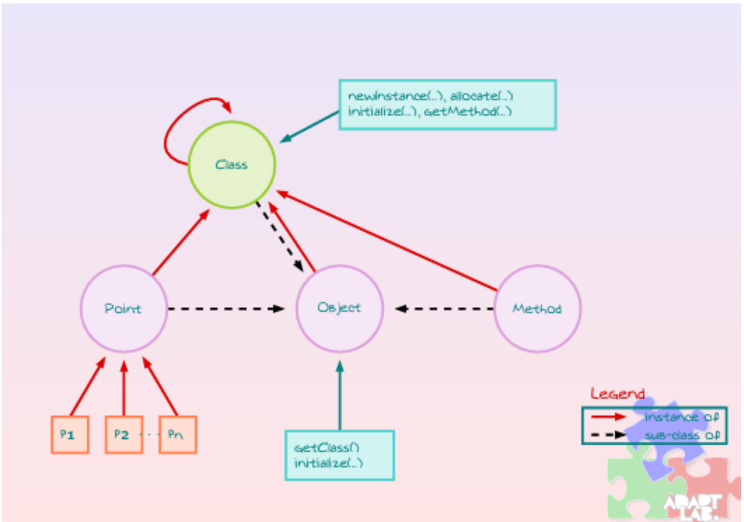
3.2.1 Es.To Enrich the Behavior of a Method Call



3.2.2 Different views



3.2.3 Classes as meta objects



3.2.4 Classes AS Meta-Objects (Cont'd)

The meta-class based approach

- the classes carry out there flective activity
- the reflective tower is realized by the inheritance link

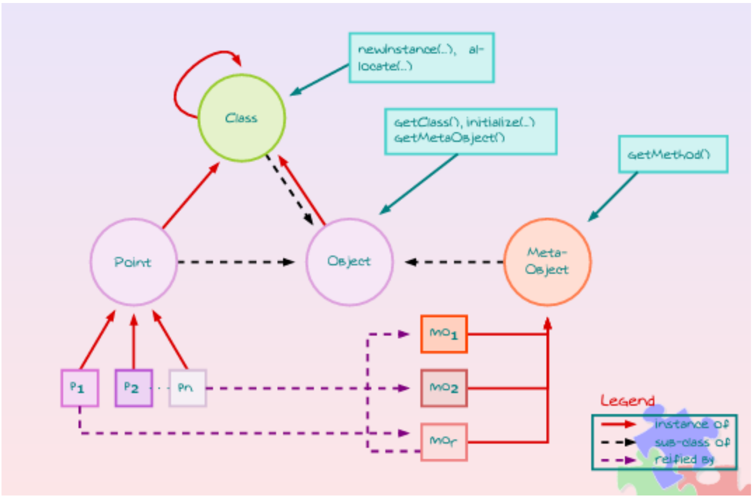
Drawbacks

- all the instances of a classs hare the same meta-class therefore the same reflective behavior (the granularity of reflection is at the class level)
- the classes have to be available a trun-time

Programming Languages

- SmallTalk (Adele Goldberg,1972)
- ObjVLisp (PierreCointe,1987)
- IBMSystemObjectModel (IBM,1992)

3.2.5 Classes AND meta objects



3.2.6 Classes AND Meta-Objects (Cont'd)

The meta-class based approach

- some special objects instantiated by a special class are associated to the base-level objects, they deal with the reflective computation
- the reflective tower is realized by clientship

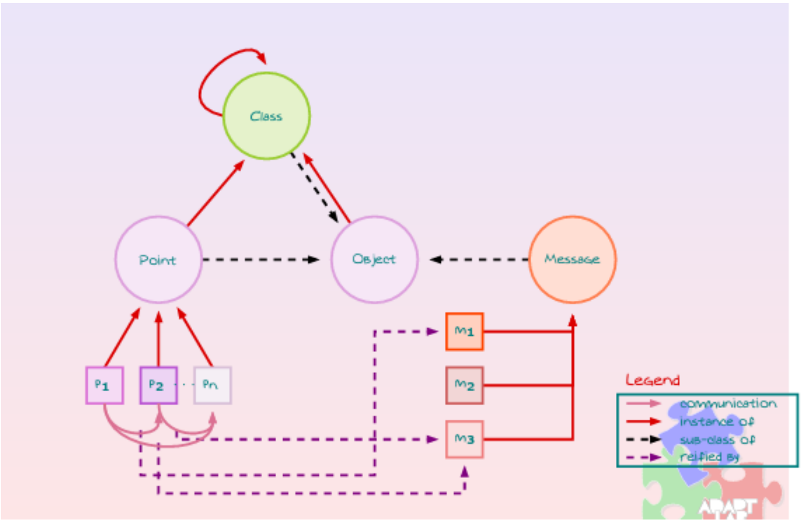
Drawback and Benefits

- the granularity of reflection is at the object level
- it cannot manage object communications, the approach lacks of a global view of the communication

Programming Languages

- CCEL(Carolyn Duby, 1992), Iguana (Brendan Gowing and Vinny Cahill, 1996);
- ABCL-R (Akinori Yonezawa and Satoshi Matsuoka, Actors meet Reflection, 1988);
- OpenC++ (Shigeru Chiba $<$ 2.0, 1993)

3.2.7 Reification of the communication



3.2.8 Classes AND Meta-Objects (Cont'd)

Approach to the reification of the communication

- some special objects reify the messages exchanged among the baselevel objects, these special objects deal with the reflective computation.

Drawback and Benefits

- the granularity of reflection is at the level of method call (very flexible)
- it is possible to reflect on the whole message exchange (global view)
- there is a meta-entities proliferation; and
- the lifecycle of the meta-entities is strictly tied to the lifecycle of the message exchange (lost the history of the reflective computation)

Programming Languages

- Mering (Jacques Ferber, 1987)
- CodA (Jeff McAffer, 1994), mChaRM (Walter Cazzola, 2001)

3.2.9 Conclusion

Computational Reflection

- It permits to open up a system to postpone some decisions - the same philosophy adopted by the late-binding mechanism.
- it depends on the awareness that a system have of itself - strictly related to the “self” of the object-oriented programming languages
- it specializes some of the object-oriented basic mechanisms (constructors, invocations, and so on) - it exploits the classic mechanisms: inheritance, delegation

Its use produces a better comprehension of the object-oriented mechanism and of their implementation

4 Meta-object Protocol and Separation of concerns

4.1 Open Implementation & Meta-Object Protocol

4.1.1 Introduction

"The work presented in this book is based ona **simple intuition**:

if **substrate systems** like programming languages, object the details of the implementation of the base-level system are open up to the meta-level system. systems, databases or operating systems can **be tailored** to

meet particular application needs as they arise,

rather than having **to hack around** existing deficiencies,

application writers are better of."

Cit. Gregor Kiczales and Andreas Pæpcke

4.1.2 System Awareness

The computational reflection allows a system of observing and manipulating its components

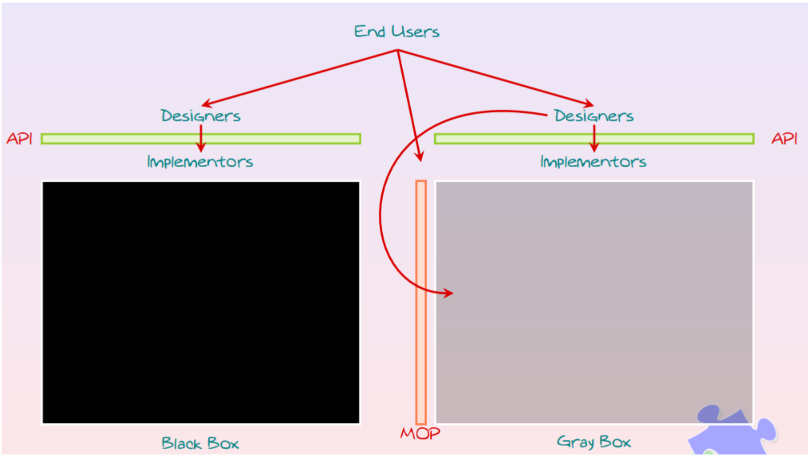
In particular

- the meta-level entities observe and manipulate the base-level entities, and
- these are NOT aware of being observed and manipulated.

Therefore:

- there is a "black-box" use of the functionality of the base-level system
- the behavior of the base-level system and its structure can be dynamically modified, and
- the details of the implementation of the base-level system are open up to the meta-level system

4.1.3 Black- and Gray-Box Approaches



4.1.4 Black- and Gray-Box Approaches (Cont'd)

Black box

- the accesses to the system functionality is limited to the mechanisms provided by the adopted programming language
- an attempt of using the system functionality can raise an "application mismatch" when a component is used in the wrong way;
- flexibility is really limited

Gray Box

- open implementation
- the component behavior can be adapted to our needs
- we can bypass the mechanisms provided by the programming language to access the system functionality
- we can re-class the objects respecting their use and behavior

4.1.5 Kinds of Opening

(At least) 3 ways to open up the system details are possible:

- **introspection**, is the system ability of observing the state and the structure of the system itself
- **intercession**, is the system ability of modifying the behavior and the structure of the system itself;
- **invoke**, is the system ability of applying the system functionality

4.1.6 Examples of MOP

Non-typed and interpreted programming languages

- Lisp - CLOS (Gregor Kiczales, 1991), ObjVLisp (Pierre Cointe, 1987), ABCL-R (Akinori Yonezawa, 1988)

Typed and interpreted programming languages

- Java - java.lang.reflect (Sun, 1995) - OpenJava (Michiaki Tatsubori, 1999), Javassist (Shigeru Chiba, 2000), Reflex (Eric Tanter, 2001).

Compiled programming languages

- C/C++ - OpenC++ (Shigeru Chiba, 1993-1995), SOM/DSOM (Ira Forman, 1994), Iguana (Vinny Cahill, 1996).

4.2 Separation of Concerns (SoC)

4.2.1 Introduction

Complete Application

=

Core Functionality

(e.g., banking applications: accounts, clients, operations, ...)

+

Nonfunctional Concerns

(security, persistence, distribution, exception handling, concurrency, ...)

Note that the separation between functional and nonfunctional is not so clear and neat.

4.2.2 Introduction (Cont'd)

Traditionally

- separation of concerns is at design stage only
- source code is a mix of all concerns (functional and nonfunctional) - error prone - bad reusability and extensibility

SoC aims at enabling such a separation in the implementation

- reflection, aspect-oriented programming

4.2.3 Separation of Concerns Get as Reflective Activity

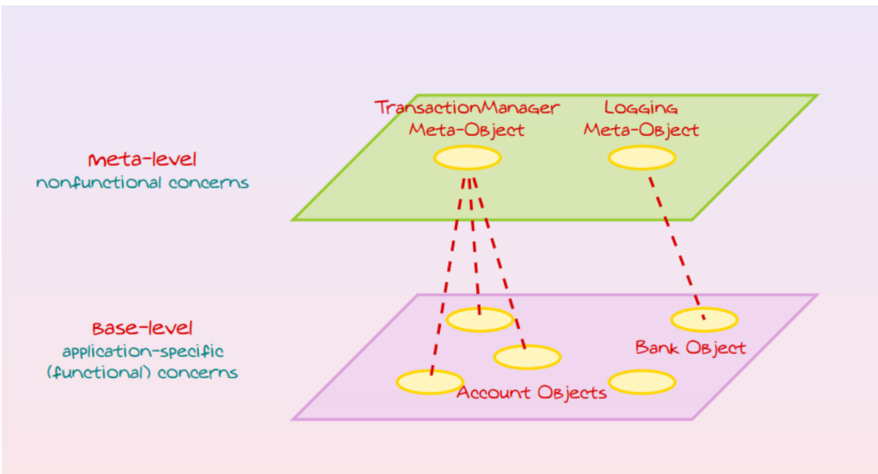
Reflection allows the designer of separating the functional aspects from the nonfunctional ones

Therefore, we get [*]:

- an augmentation of the functionality reuse
- an augmentation of the system stability; and
- the functional and nonfunctional aspects can be developed independently

[*] Walter Hürsch and Cristina Videira-Lopes. Separation of Concerns. TR. NU-CCS-95-03 Northeastern University. February 1995.

4.2.4 Separation of Concerns Get as Reflective Activity (Cont'd)



5 Java Reflection

5.1 Reflection in Java

5.1.1 Introduction

The **Java Core Reflection API** provides a small, type-safe, and secure API that supports introspection about the classes and objects in the current Java Virtual Machine.

If permitted by security policy, the API can be used to:

- construct new class instances and new array
- access and modify fields of object and classes
- invoke methods on object and classes, and
- access and modify elements of array

Intercession on classes and objects is forbidden

5.1.2 Introduction (Cont'd)

The Java application that benefint from introspection are:

- automatic documentation (javac, javadoc, ...)
- tools for IDEs: browsers, inspectors, debuggers, ...
- serialization / deserialization - construction of a binary representation for backup or transmission; - re-creating an object based on its serialized form;
- RMI - serialization of arguments and return values; - identification of remote methods.

5.1.3 Classes and Interfaces for Reflection

Since Java $<$ 12

- java.lang.Object
  - java.lang.Class
  - java.lang.reflect.Member
    java.lang.reflect.Field (Member)
    java.lang.reflect.Method (Member)
    java.lang.reflect.Constructor (Member)

Since Java 13

- java.lang.Object
  - java.lang.Class
  - java.lang.reflect.Member
  - java.lang.reflect.AccessibleObject
    java.lang.reflect.Field (Member)
    java.lang.reflect.Method (Member)
    java.lang.reflect.Constructor (Member)
- java.lang.reflect.Proxy
- java.lang.reflect.InvocationHandler

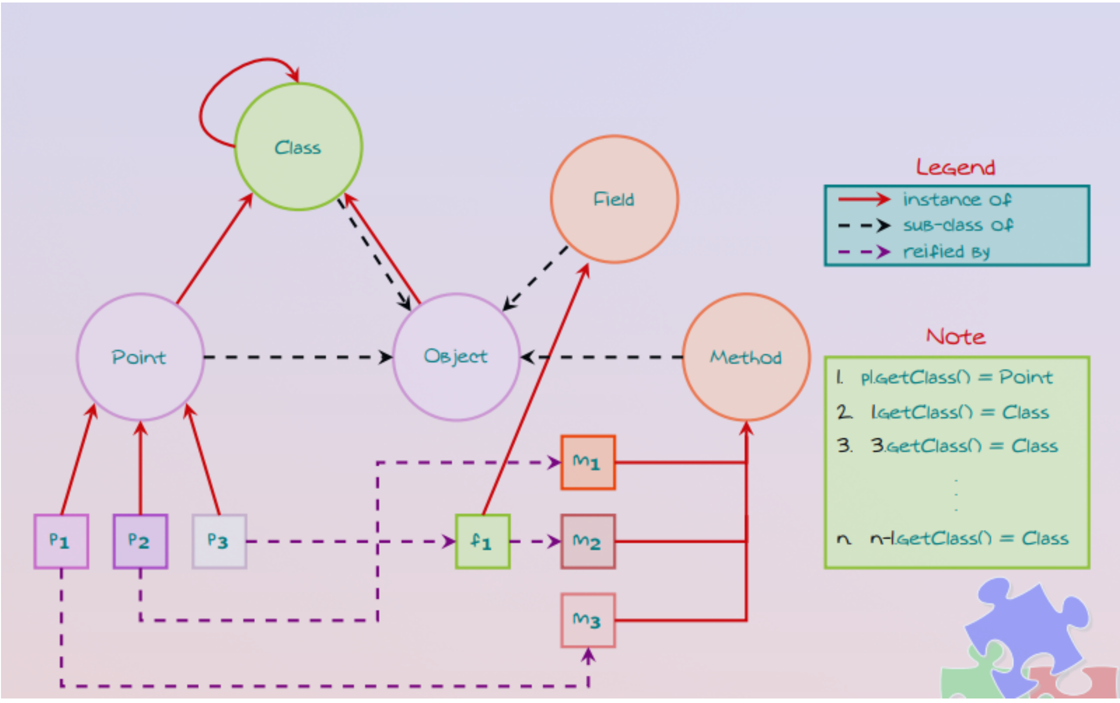
- boolean.class, char.class, int.class, double.class, ...

Since Java 15

- java.lang.Object
  - java.lang.Class
  - java.lang.reflect.Member
  - java.lang.reflect.AccessibleObject
    java.lang.reflect.Field (Member)
    java.lang.reflect.Method (Member)
    java.lang.reflect.Constructor (Member)
- java.lang.reflect.Proxy
- java.lang.reflect.InvocationHandler
- java.lang.annotation.Annotation
- java.lang.instrument.Instrumentation

boolean.class, char.class, int.class, double.class, ...

5.1.4 The Java's Class Model



5.1.5 Java's Limitations on Reflaction

The meta-object protocol is no causally connected - Causal connection-poses a security rick, which has not been analyzed in the context of Java's bytecode verifier

Class is declared final - One cannot create new meta-classes

There are no MOP operations to modify classes) - Therefore, one cannot easily create and modularize class-to-class transformation

Do the Java designers disagree with such transformtations? No

5.2 Java Reflection API (Package java.lang.reflect)

5.2.1 Class-to-Class Transformations: Marker Interfaces

Consider

- Clonable
- Remote

- Serializable

Are these really interfaces? No

If not, what are they? - built-in class-to-class transformations

Java programmers cannot directly create such transformations

Other techniques must be employed ... - Some of them are the subject of the rest of this course

5.2.2 Methods of Object

Object defines method to which all objects respond

```
class Object {
  public final Class<?> getClass() { ... }
  protected Object clone() { ... }
  public boolean equals(Object obj) { ... }
  public int hashcode() { ... }
  public String toString() { ... }
  public final void notify() { ... }
  public final void notifyAll() { ... }
  public final void wait() { ... }
    ...
}
```

5.2.3 Methods of Class<T>

Methods of Class<T>—Basic Operations

```
public final class Class<T> extends Object {
  public static Class<?> forName(String className) { ... }
  public static Class<?> forName(Module module, String name) { ... }
  public T newInstance() { ... } /* deprecated since 9 */
  public boolean isInstance(Object obj) { ... }
  public String getName() { ... }
  public Class<? super T> getSuperclass() { ... }
  public Module getModule() { ... }
  public Class<?>[] getInterfaces() { ... }
  public Class<?>[] getDeclaredClasses() throws SecurityException { ... }
  public Method[] getDeclaredMethods() throws SecurityException { ... }
  public Constructor<?> getEnclosingConstructor()
      throws SecurityException { ... }
  public Field[] getFields() throws SecurityException { ... }
    ...
}
```

5.2.4 java.lang.Class at Work

Let's write a method to return a printable class name

```
class MOP {
  public static String classNameToString(Class<?> cls) {
    if (!cls.isArray()) return cls.getName();
    else return cls.getComponentType().getName() + "[]";
  }
}
```

```
[14:40]cazzola@hymir:~/tsp>jshell
| Welcome to JShell -- Version 11
| For an introduction type: /help intro
jshell> /open MOP1.java
jshell> MOP.classNameToString(String.class)
$2 ==> "java.lang.String"
jshell> var a = new Integer[]{1,2,3}
a ==> Integer[3] { 1, 2, 3 }
jshell> a.getClass()
$4 ==> class [Ljava.lang.Integer;
jshell> MOP.classNameToString(a.getClass())
$5 ==> "java.lang.Integer[]"
jshell> /exit
| Goodbye
```

Let's code a method to return super class hierarchy of a class

```
import java.util.ArrayList;
import java.util.List;

class MOP {
  public static Class<?>[] getAllSuperClasses(Class<?> cls) {
    List<Class<?>> result = new ArrayList<Class<?>>();
    for (Class<?> x = cls; x != null; x = x.getSuperclass())
      result.add(x) ;
    return result.toArray(new Class<?>[0]);
  }
}
```

```
[16:33]cazzola@hymir:~/tsp>jshell
jshell> /open MOP2.java
jshell> MOP.getAllSuperClasses(java.util.ArrayList.class)
$4 ==> Class[4] { class java.util.ArrayList, class java.util.AbstractList,
                  class java.util.AbstractCollection, class java.lang.Object }
```

5.2.5 Summary for Class<T> Methods

Member Access

getAnnotations

getAnnotation

getClasses

getConstructors

getConstructor

getDeclaredAnnotation

getDeclaredClasses

getDeclaredConstructors

getDeclaredConstructor

getDeclaredFields

getDeclaredField

getDeclaredMethods

getDeclaredMethod

getFields

getField

getMethods

getMethod

Class Properties

getComponentType

getDeclaringClass

getEnclosingClass

getEnclosingConstructor

getEnclosingMethod

getModifiers

isAnnotationPresent

isAnnotation

isAnonymousClass

isArray

isAssignableFrom

isEnum

isInterface

isPrimitive

Context Access

getClassLoader

getInterfaces

getModule

getPackage

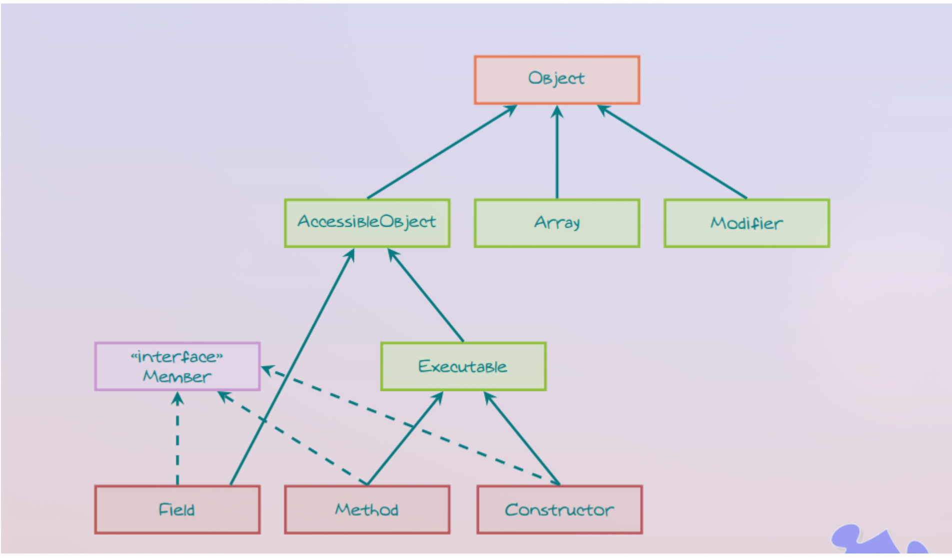
getProtectionDomain

getResourceAsStream

getResource

getSigners

5.2.6 Classes in java.lang.reflect



Instances of Field, Constructor, and Method are meta-objects

5.2.7 java.lang.reflect.Method

```
public final class Method extends Executable implements Member {
  public Class<?> getReturnType() { ... }
  public Class<?>[] getParameterTypes() { ... }
```

```java
  public Class<?>[] getExceptionTypes() { ... }
  public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
      InvocationTargetException { ... }
      ...
}
```

5.2.8 When using invoke:

- individual parameters are automatically unwrapped to match primitive formal parameters, and
- both primitive and reference parameters are subject to method invocation conversions as necessary.

The return type is automatically wrapped in an Object

```java
import java.util.stream.*;
import java.util.Arrays;
import java.lang.reflect.Method;

class MOP {
  public static String headerSuffixToString(Method m) {
    String result = MOP.classNameToString( m.getReturnType() )
        + " " + m.getName()
        + "(" + MOP.formalParametersToString( m ) + ")";
    Class<?>[] exs = m.getExceptionTypes();
    if (exs.length > 0)
      result += " throws " + MOP.classArrayToString(exs);
    return result;
  }
}
```

```
[20:28]cazzola@hymir:~/tsp>jshell
jshell> /open MOP4.java
jshell> var cls = Class.forName("java.lang.reflect.Method")
cls ==> class java.lang.reflect.Method
jshell> var ms = Arrays.asList(cls.getDeclaredMethods()).stream()
  .filter(s->s.getName()=="invoke")
ms ==> java.util.stream.ReferencePipeline$2@548e7350
jshell> ms.forEach(m -> System.out.println(MOP.headerSuffixToString(m)))
java.lang.Object invoke(java.lang.Object p1, java.lang.Object[] p2)
  throws java.lang.IllegalAccessException,
      java.lang.IllegalArgumentException,
  java.lang.reflect.InvocationTargetException
```

5.2.9 java.lang.reflect.Field

```java
public final class Field extends AccessibleObject implements Member {
  public Class<?> getType() { ... };
  public Object get(Object obj)
    throws IllegalArgumentException, IllegalAccessException { ... };
  public void set(Object obj, Object value)
    throws IllegalArgumentException, IllegalAccessException { ... };
  public Class<?> getDeclaringClass() {...} ;
    ... // Include get* and set* for the eight primitive types
}
```

5.2.10 java.lang.reflect.AccessibleObject

Purpose It is the base class for Field, Method and Constructor objects - In this last two cases inherited by the Executable class.

It enables the suppression of the access control checks when:

- setting or getting fields (using Field)
- invoking methods (using Method)
- creating and initializing new instances of classes (Constructor)

```java
public final class AccessibleObject {
  public void setAccessible(boolean flag) throws SecurityException { ... }
  public static void setAccessible(AccessibleObject[] array, boolean flag)
      throws SecurityException { ... }
  public boolean isAccessible() { ... }
}
```

Note that the Java security manager can forbid the use of setAccessible()

5.2.11 java.lang.reflect.AccessibleObject (Cont'd)

```java
import java.lang.reflect.Field;

class Employee {
  private String name;
  public Employee(String name) { this.name = name; }
}

class AccessibilityCheck {
  public static void main(String[] args) {
    try {
      Employee mike = new Employee("Mike");
      Field name = Employee.class.getDeclaredField("name");
      name.setAccessible(true);
```

```
      System.out.println("Value of name: "+ name.get(mike));
      name.set(mike, "Eleonor");
      System.out.println("Changed value of name: " + name.get(mike));
    } catch(NoSuchFieldException | SecurityException |
        IllegalAccessException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

```
grant {
  permission java.lang.reflect.ReflectPermission
  "suppressAccessChecks";
};
```

```
[22:53]cazzola@hymir:~/tsp>java AccessibilityCheck
Value of name: Mike
Changed value of name: Eleonor
[23:02]cazzola@hymir:~/tsp>java -Djava.security.manager AccessibilityCheck
access denied ("java.lang.reflect.ReflectPermission" "suppressAccessChecks")
[23:03]cazzola@hymir:~/tsp>java -Djava.security.policy=granted.policy
                               -Djava.security.manager AccessibilityCheck
Value of name: Mike
Changed value of name: Eleonor
```

5.2.12 java.lang.reflect.Constructor

```
public final class Constructor extends AccessibleObject implements Member {
  public T newInstance(Object... initargs)
    throws InstantiationException, IllegalAccessException,
      IllegalArgumentException, InvocationTargetException { ... }
  ...
}
```

Note that newInstance() of Class invokes default constructor, other constructor are invoked with newInstance() of Constructor

5.2.13 Examples: Smart Reflective Access to Fields

```
import java.lang.reflect.*;

public interface SmartFieldAccess {
  default public Object instVarAt(String name) throws Exception {
    Field f = this.getClass().getDeclaredField(name);
    f.setAccessible(true);
    if (!Modifier.isStatic(f.getModifiers())) return f.get(this);
    return null;
}
```

```java

default public void instVarAtPut(String name, Object value)
      throws Exception {
  Field f = this.getClass().getDeclaredField(name);
  f.setAccessible(true);
  if (!Modifier.isStatic(f.getModifiers())) f.set(this, value);
  }
}

class Employee implements SmartFieldAccess {
  private String name;
  public Employee(String name) {this.name=name;}
}
```

```
[0:24]cazzola@hymir:~/tsp>jshell
jshell> /open SmartFieldAccess.java
jshell> var mike = new Employee("Mike");
mike ==> Employee@59f99ea
jshell> mike.instVarAtPut("name", "Eleonor")
jshell> mike.instVarAt("name")
$6 ==> "Eleonor"
```

5.2.14 Examples: Reflective Cloning

```java
import java.lang.reflect.Field;

public interface ReflectiveCloning {
  default public Object copy() throws Exception {
    Object tmp = this.getClass().getDeclaredConstructor()
           .newInstance() ;
    Field[] fields = this.getClass().getDeclaredFields();
    for (int i = 0; i < fields.length; i++) {
      fields[i].setAccessible(true);
      fields[i].set(tmp, fields[i].get(this));
    }
    return tmp;
  }
}

class Employee implements ReflectiveCloning {
  private String name;
  public Employee() {this.name="Anon"; }
  public Employee(String name) {this.name=name;}
  public String toString() {return "Employee: "+this.name;}
}
```

```
[1:04]cazzola@hymir:~/tsp>jshell
jshell> /open ReflectiveCloning.java
jshell> var e = new Employee("Mike");
```

```
e ==> Employee: Mike
jshell> var e1 = e.copy();
e1 ==> Employee: Mike
```

5.2.15 Examples: Reflective Cloning

```
import java.lang.reflect.Method;

public interface SmartMessageSending {
  default public Object receive(String selector, Object[] args)
        throws Exception {
    Method mth = null; Class<?>[] classes = null;
    if (args != null) {
      classes = new Class<?>[args.length];
      for (int i = 0; i < args.length; i++) classes[i] = args[i].getClass();
    }
    mth = this.getClass().getMethod(selector, classes);
    return mth.invoke(this, args);
  }
}

class Employee implements SmartMessageSending {
  private String name;
  public Employee(String name) {this.name=name;}
  public void setName(String name) {this.name=name;}
  public String getName() {return this.name;}
  public String toString() {return "Employee: "+this.name;}
}
```

```
jshell> var e = new Employee("Mike");
e ==> Employee: Mike
jshell> e.receive("getName", null)
$1 ==> "Mike"
jshell> e.receive("setName", new Object[]{"Eleonor"})
$2 ==> null
jshell> e
e ==> Employee: Eleonor
```

5.3 Conclusions

Benefits

- reflection in Java opens up the structure and the execution trace of the program
- the reflective API is simple and quite complete

Drawbacks

- reflection in Java is limited to introspection
- there isn't a clear separation between the two logical layers (baseand meta-level)
- reflection in Java has been proved inefficient