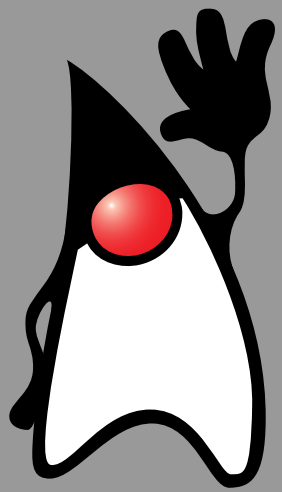
EPITA Première Année Cycle Ingénieur
Atelier Java - J3

Marwan Burelle

marwan.burelle@lse.epita.fr
http://www.lse.epita.fr



Overview

Plan

1. Imagination From The Other Side
   - Reflection ?
   - Getting Class
   - Architecture of Reflexivity in Java

Plan

What is Reflection ?

Reflection is the process by which a computer program can observe and modify its own structure and behavior.

- Reflective language should be able of **introspection**
- Using reflective language, we should be able to call functions, methods or whatever, without direct access (for example, be able to call a function using a string representing its name.)

Example:

```
public static <T> void inspect(T t){
  System.out.println(t.getClass().getName());
}
public static <T> Object call(T t, String fun){
  Method m = t.getClass().getMethod(fun);
  return (m.invoke(t));
}
```

Why Reflection ?

- Serialization/Unserialization
- Auto-adaptative programs
- Metaprogrammation
- Calling entity dynamicaly by their name

Example:

```
public static <T> Object call(T t, String fun,
                              Class<?>[] paramTypes,
                              Object[] params)
  throws NoSuchMethodException,
         IllegalAccessException,
         InvocationTargetException
{
  Method m = t.getClass().getMethod(fun,paramTypes);
  return (m.invoke(t,params));
}
```

Reflection in Java

- Java's objects integrate type meta-information
- Class definitions (and almost any definitions) are represented using Java's value (objects)
- We are able to test object's class, to retrieve the inner representation of almost any entities in the language (classes, methods, fields, annotations . . .)
- We are able to **use** or **create** entities using their inner representation

Testing the class

- The operator `instanceof` let you test the class of an object
- If object `o` belongs to class `C`, then `o instanceof C` will return `true`
- `o instanceof C` will return true if `o` was instantiate from:
  - class `C`
  - a subclass of `C`
  - a class implementing interface `C`
- `instanceof` can not test for generic type:
  - you can **not** write `o instanceof List<C>`
  - you must write `o instanceof List`

Example:

```
import java.util.*;
import static java.lang.System.out;

class Elm {
    private String name;
    public String getName() { return name; }
    public Elm(String n) { name = n; }
}

class IntElm extends Elm {
    private int id;
    public IntElm(int n) {
        super("id_" + n);
        id = n;
    }
    public int getId() { return id; }
}
```

Example:

```
public class BasicIntrospection {
  static public void printOnlyIntElm(List<Elm> l) {
    for (Elm e : l) {
      if (e instanceof IntElm)
        out.format("Elm id: %d%n",
          ((IntElm)e).getId());
    }
  }
}
```

Plan

The getClass Method

Example:

```
Class c = "foo".getClass();
System.out.println(c);
```

class java.lang.String

Example:

```
public class MyReflectClass {
  public void autoInspect(){
    System.out.println(this.getClass());
  }
  public static void main(String[] args){
    MyReflectClass o = new MyReflectClass();
    o.autoInspect();
  }
}
```

class MyReflectClass

Getting Class From Class

Example:

```
// Accessing Class object from a class
Class c = MyReflectClass.class;
// Or from an atomic type
Class boolClass = boolean.class;
System.out.println(c);
System.out.println(boolClass);
```

```
class MyReflectClass
boolean
```

The `.class` notation is the only way to acces the class of a native type (you can't dereference a variable of native type.) You can also get the class of a native type from the wrapping class:

```
System.out.println(Boolean.TYPE);
```

```
boolean
```

Getting Class From its Name

If we have the *fully-qualified name* we can get the corresponding `Class` object:

Example:

```
Class sc = Class.forName("java.lang.String");
System.out.println(String.class == sc);
```

true

Example:

```
public Foo {
  static Object getObjFromName(String className) {
    return (Class.forName(className).newInstance())
  }
}
```

Other Ways of Getting Classes

Some methods from `java.lang.Class`:

```
public Class<? super T> getSuperclass();

public Class<?>[] getClasses();

public Class<?>[] getDeclaredClasses();

public Class<?> getEnclosingClass();
```

We also have the method `getDeclaringClass()` which can be called from `Class` or `Fields` or ...

Plan

Global organisation

How to access the inner content

Member	Class API	List	Inherited	Private
Field	getDeclaredField	no	no	yes
	getField	no	yes	no
	getDeclaredFields	yes	no	yes
	getFields	yes	yes	no
Method	getDeclaredMethod	no	no	yes
	getMethod	no	yes	no
	getDeclaredMethods	yes	no	yes
	getMethods	yes	yes	no
Constructor	getDeclaredConstructor	no	-	yes
	getConstructor	no	-	no
	getDeclaredConstructors	yes	-	yes
	getConstructors	yes	-	no

Member API

- To represent members of a class we have a generic interface `java.lang.reflect.Member`;
- `java.lang.reflect.Field`: implements `Member` for properties;
- `java.lang.reflect.Method`: implements `Member` for methods;
- `java.lang.reflect.Constructor`: implements `Member` for constructors;
- Each implementation has some specific operations such as setting a fields or invoking a method.

Plan

Plan

ClassSpy: A Class Inspector

- ClassSpy goals is to extract all informations from a given class;
- We will use most of the reflection API on class;
- It will introduce us with Member API;
- This example is directly extract from the Java Tutorial: http://java.sun.com/docs/books/tutorial/reflect/class/classMembers.html

ClassSpy: preamble

Example:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Member;
import static java.lang.System.out;

enum ClassMember{CONSTRUCTOR,FIELD,METHOD,CLASS,ALL}
```

ClassSpy: printClass

Example:

```
public class ClassSpy {
 private static Class<?> printClass(String cname)
  throws ClassNotFoundException
 {
  Class<?> c = Class.forName(cname);
  out.format("Class:%n %s%n%n",c.getCanonicalName());

  Package p = c.getPackage();
  out.format("Package:%n  %s%n%n",
   (p != null ? p.getName() : "-- No Package --"));
  return c;
 }
}
```

ClassSpy: printClasses

Example:

```
private static void printClasses(Class<?> c) {
  out.format("Classes:%n");
  Class<?>[] clss = c.getClasses();
  for (Class<?> cls : clss)
    out.format("  %s%n", cls.getCanonicalName());
  if (clss.length == 0)
    out.format(
     "--No member interfaces,classes,or enums--%n");
  out.format("%n");
}
```

ClassSpy: printMembers

Example:

```
private static void printMembers(Member[] mbrs,
                                 String s) {
 out.format("%s:%n", s);
 for (Member mbr : mbrs) {
  if (mbr instanceof Field)
   out.format(" %s%n",
              ((Field)mbr).toGenericString());
  else if (mbr instanceof Constructor)
   out.format(" %s%n",
              ((Constructor)mbr).toGenericString());
  else if (mbr instanceof Method)
   out.format(" %s%n",
              ((Method)mbr).toGenericString());
 }
 if (mbrs.length == 0)
  out.format(" -- No %s --%n", s);
 out.format("%n");
}
```

ClassSpy: main

Example:

```
public static void main(String... args) {
 try {
  Class<?> c = printClass(args[0]);
  for (int i = 1; i < args.length; i++) {
   switch (ClassMember.valueOf(args[i])) {
    case ALL:
     printMembers(c.getConstructors(),"Constuctors");
     printMembers(c.getFields(), "Fields");
     printMembers(c.getMethods(), "Methods");
     printClasses(c);
     break;
    default:
     assert false;
   }
  }
 } catch (ClassNotFoundException x) {
          x.printStackTrace();
 }
}
```

ClassSpy : result

```
> java ClassSpy HelloWorld ALL
Class:
  HelloWorld

Package:
  -- No Package --

Constuctors:
  public HelloWorld()

Fields:
  -- No Fields --

Methods:
  public static void HelloWorld.main(java.lang.String[])
  public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
  public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
  public final void java.lang.Object.wait() throws java.lang.InterruptedException
  public native int java.lang.Object.hashCode()
  public final native java.lang.Class<?> java.lang.Object.getClass()
  public boolean java.lang.Object.equals(java.lang.Object)
  public java.lang.String java.lang.Object.toString()
  public final native void java.lang.Object.notify()
  public final native void java.lang.Object.notifyAll()

Classes:
  -- No member interfaces, classes, or enums --
```

Plan

Another Use Case

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

public class View {
    private Object engine;
    public void addEngine(Object e) {
        engine = e;
    }
    private Object engineGetRes()
        throws NoSuchMethodException,
               IllegalAccessException,
               InvocationTargetException
    {
        Method getRes = engine.getClass().getMethod("getRes");
        return (getRes.invoke(engine));
    }
    public void start()
        throws NoSuchMethodException,
               IllegalAccessException,
               InvocationTargetException
    {
        System.out.println("Welcome");
        System.out.println("Res: "+ this.engineGetRes());
    }
}
```

Another Use Case

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

public class Engine {
    private Object view;

    public void addView(Object v) { view = v; };
    private int fact(int i, int accu) { return (i>0?fact(i-1,accu*i):accu); }
    public Integer getRes() { return (new Integer(fact(5,1))); }

    public void startup()
        throws NoSuchMethodException,
               IllegalAccessException,
               InvocationTargetException
    {
        Method s = view.getClass().getMethod("start");
        s.invoke(view);
    }

    public static void main(String[] args) {
        Engine engine = new Engine();
        try {
            View view = new View();
            view.addEngine(engine);
            engine.addView(view);
            engine.startup();
        }
        catch // [snip error handling here]
    }
}
```

Plan

Threads in Java

- As every modern languages, Java offers multi-threading facilities
- One can instantiate directly `Thread` object to create new threads:
  - by extending the class `Thread`
  - by implementing the interface `Runnable`
- One can use high level `Executor`
- One can use *Thread Pools*

Plan

Extending Thread

Example:

```
public class MyThread extends Thread {
 public void run() {
  for (int i=0; i<100; i++) {
   System.out.println(
           Thread.currentThread().getName()
         + ": Hello !");
   try {
    Thread.sleep(400);
   } catch (InterruptedException e) {}
  }
 }
 public static void main(String... args) {
  (new MyThread()).start();
  (new MyThread()).start();
 }
}
```

Plan

The Runnable Interface

- Runnable interface provide an abstraction to be used with a thread and still can extend a class other than Thread
- What you have to do is to provide a run method which will be called when the surrounding thread will start

Example:

```
public class FooBar implements Runnable {
 void run(){
  System.out.println("A Runnable Object");
 }
 public static void main(String[] args){
  (new Thread(new FooBar())).start()
 }
}
```

Implementing Runnable

Example:

```
public class MyRunnable implements Runnable {
 public void run() {
  for (int i=0; i<100; i++) {
   System.out.println(
           Thread.currentThread().getName()
         + ": Hello !");
   try {
          Thread.sleep(400);
   } catch (InterruptedException e) {}
  }
 }
 public static void main(String... args) {
  (new Thread(new MyRunnable())).start();
  (new Thread(new MyRunnable())).start();
 }
}
```

Plan

Plan

What is a monitor

- Monitor is the object oriented way of dealing with mutual exclusion.
- Normally a monitor is an object such that **only one** thread can execute code *in the monitor* at a time.
- A monitor also provides a condition mechanism to put thread in sleep until a specific event.
- There are two kind of condition mechanism:
  - Blocking (*Hoare style*): the signaled thread take precedence over the signaling thread.
  - Nonblocking (*Mesa style*): the signaling thread continue execution and the signaled thread is just put on the waiting queue (with eventually some priority over newer threads.)
- In Java all objects are monitors.

Using Condition

- A condition variable is an abstract entity associated (rather implicitly) with a condition on a part of the monitor.
- We use condition variable with *two roles*:
  - *waiting*: when a thread need for the condition to be met (true), it will wait on the specific condition variable;
  - *signaling*: (sometimes called *notifying*) when a thread modify the state of the object so that the condition can be met, it will signal it to the waiting threads (one or all.)
- As stated earlier, blocking condition semantics will give hands to the newly waked-up threads, while nonblocking condition semantics will just push waiting threads on the scheduling queue of the monitor. When using nonblocking condition, we usually talk about *notifying* rather than *signaling*.

Using Conditions

- The condition variable doesn't describe the condition by itself, but the state (change or not) of the condition. So, the basic scheme of using condition variables is as follow:

```
// waiting thread
{
  // Use loop rather than "if" statement here,
  // you can be waked for other reason than
  // what you're waiting for.
  while (P) do wait(C);
  // do what you need: you're in the monitor again
}
// signaling thread
{
  // do some work
  signal(C);
  // ...
  // leaving the monitor
}
```

Java's monitor

- Every object are monitor, code (or method) in mutual exclusion must be explicitly marked using `synchronized` modifiers on method or `synchronized` blocks.
- Condition variables are *implicit*: *wait* and *notify* operations apply to all threads in the monitor.
- Java's monitor can be used for code in other objects (thus acting as simple lock or condition variables.)
- The condition mechanism is a *nonblocking* one.
- Basic locking and synchronization do not require specific entity (such as lock) since the base class `Object` provides all what we need.

Plan

Synchronized Methods

Example:

```
public class SyncCount {
 private int c;
 public SyncCount(int start) { c = start ;}
 public synchronized void incr(){ c++; }
 public synchronized void decr(){ c--; }
 public synchronized int value(){ return c; }
}
```

Synchronized Blocks

Example:

```
public class DualSyncCount {
 private int c1;
 private int c2;
 private Object lock1 = new Object();
 private Object lock2 = new Object();
 public void add1() {
  synchronized(lock1) {
   c1++;
  }
 }
 public void add2() {
  synchronized(lock2) {
   c2++;
  }
 }
}
```

Plan

Waiting and notifying

- Java doesn't offer condition variables (such as `pthread_cond_wait` and `pthread_cond_signal`) but offers a similar mechanism in the object monitor.
- When a thread invoke the method `wait` of an object, it will be put to sleep.
- When a thread invoke the method `notify` (or `notifyAll`) on an object it will schedule back a thread waiting on the same object (or all waiting threads.)
- Since *waiting* and `notifying` can be called on any object (not only `this`) one can used specific object to simulate condition variables.

Producer/Consumer

```
public class Prod<E> {
  private Queue<E> q;
  public void produce(E data) {
    synchronized (q) {
      q.add(data);
    }
    q.notifyAll();
  }
}
public class Conso<E> {
  private Queue<E> q;
  public E consume() {
    synchronized (q) {
      while (q.isEmpty()) {
        try { q.wait(); }
        catch (InterruptedException x) {}
      }
      return q.peek();
    }
  }
}
```

Plan

A Quick Survey On Semaphore

- Semaphores are abstract entity used to synchronized threads
- They comes with an inner counter
- They provide two operations: P (get) and V (set)
- P wait until the counter becomes positives and thus decrease it
- V increase the counter

First Attempt

Example:

```
public class Semaphore {
 private int count;
 public Semaphore() { count = 1; }
 public Semaphore(int start) { count = start ; }

 // won't work: we never release the monitor.
 public synchronized void P(){
  while (count==0){}
  count--;
 }
 public synchronized void V(){
  count++;
 }
}
```

Synchronized Blocks

Example:

```
public class Semaphore {
 private int count;
 public Semaphore() { count = 1; }
 public Semaphore(int start) { count = start ; }

 public void P(){
  while (count==0){}
  synchronized (this) {
   count--;
  }
 }
 public synchronized void V(){
  count++;
 }
}
```

Using wait

Example:

```
public class Semaphore {
 private int count;
 public Semaphore() { count = 1; }
 public Semaphore(int start) { count = start ; }

 public synchronized void P(){
  while (count==0){
   try {
    wait();
   } catch (InterruptedException e) {}
  }
  count--;
 }
 public synchronized void V(){
  count++;
  notifyAll();
 }
}
```

Going Further ...

- Semaphores use a waiting queue and thus easily satisfy fairness
- It will be good to use a similar mechanism
- `notify` and `notifyAll` does not offers a mechanism to wake a specific thread.
- We may use `Thread.interrupt`
- Nevertheless there is a class Semaphore: java.util.concurrent.Semaphore

More threads ?

- Java offers many *thread safe* collections
- Java also offers a kind of *Pool of threads* for cases where many threads are needed.
- There are also *true* mutex lock.
- You can also try the pseudo functionnal approch by constructing purely functionnal data structures.

Plan

Threads Design Patterns

- Parallel and concurrent computing are not as simple as launching threads;
- Doug Lea's book "*Concurrent Programming in Java: Design Principles and Patterns*" is good starting point;
- Most principles exposed in the book are now integrated in the base API;
- You should use concurrent data structures rather than just basic locking on non concurrent one;
- Java execution models implies different issues in multi-threading so programmers should use provided tools rather than trying to port non-Java parallel code.

Example: Concurrent Queue

- Simple implementation of concurrent queue (one lock and a condition variable) induces far too much contention;
- There exists clever queue implementations:
  - Fully non-blocking queue using *double-compare-and-swap* (with a nice progression property)
  - Double locks queue
- Implementing these algorithms in Java is tedious (hey, how do we use *double-compare-and-swap* in a non native programming language ?);
- Luckily, Java implementation of concurrent queue uses the best techniques available !

Thread-friendly data structures

- Java Collections provide thread oriented data structures;
- `java.util.concurrent` provides several concurrent queues implementation with different properties;
- `ConcurrentMap` and `ConcurrentHashMap` provides thread-safe and thread-friendly map;
- Java also provides *copy-on-write* arrays:
  - *copy-on-write* data structures are *never* blocking data structures usefull when modifications are *rare* and when you need safe iterations;
  - `CopyOnWriteArraySet`: provide a `Set` implementation using *copy-on-write*;
  - `CopyOnWriteArrayList`: same as above but for `List` interface
- Java provides *Atomic Variables* that is, simple value with atomic operations (`get`, `set` ...)

Concurrent Queue

- `LinkedBlockingQueue`: an optionally bounded FIFO blocking queue backed by linked nodes;
- `ArrayBlockingQueue`: a bounded FIFO blocking queue backed by an array;
- `PriorityBlockingQueue`: an unbounded blocking priority queue backed by a heap;
- `DelayQueue`: a time-based scheduling queue backed by a heap;
- `SynchronousQueue`: a simple *rendez-vous* mechanism that uses the BlockingQueue interface.

High Level Threads

- `Executor` interface (and derived interface) provides smarter interface on threads management;
- The idea is to *decouple* inner thread management and tasks launching: the implementation choose how and when threads are launched while the interface provide a way to start new tasks (implementing `Runnable`);
- `ExecutorService` interface offers also task launching with *returned value* using `Callable` and `Future` interface;
- Most implementations provided by `java.util.concurrent` use threads pool: the executor manages a set of running threads (*worker*) and scheduled launched tasks on available worker;
- `java.util.concurrent.Executors` provides several factory methods for most needed threads pool based executors.