What is Reflection

- Reflection provides user code access to internal information (type, fields, methods, constructors etc.) of classes loaded into the JVM
- Package: `java.lang.refect`; except for the `Class` class (`java.lang`)

```
class Vegi {
     static {  System.out.println("Loading Vegi Pizza"); }
}
class Cheese {
     static {  System.out.println("Loading Cheese Pizza"); }
}
public class PizzaStore {
     public static void main(String[] args) {
          Vegi v = new Vegi();
          try {
               Class<?> c = Class.forName(args[0]);
          } catch(ClassNotFoundException e) {
               e.printStackTrace();
          }
     }
}
```

The Class class

Java's run-time type information is represented by special objects: the `Class` class object.
In a Java program, there is a `Class` object for each class.
Three ways to get a `Class` object:

- Class literals provide a reference to the Class object.

```
Class<?>  c = java.awt.Button.class;
Class<?>  c = Foo.class;
```

- By a `getClass()` method: returns the `Class` object of an `obj`.

```
Class<?> C = obj.getClass();
```

- By class name: dynamic loading of a class:

```
Class<?>  c = Class.forName(name);
```

The Class class

The `Class` class represents any Java types. (classes, interfaces, arrays and primitives)
The matching checks:

- `isInterface();`
- `isPrimitive();`
- `isArray();`
- ...

We can explore the content of a class through an Class object.

- `getMethods;`
- `getFields;`
- `getConstructors;`
- ...

Example 1: R/W Fields, Call Methods, Instantiate Classes

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

class A {
  public String course = "CS";
}
class B extends A {
  private int number = 671;
  private int year = 2013;
  public String toString () {
    return course+"-"+number+"("+year+")";
  }
  public static Object getSomeObject () {
    return new B();
  }
}
public class Reflection {
  public String dbl (String s) { return s+s; }
  void someMethod () {}
```

```
public static void main(String[] args) throws Exception {
  Object o = B.getSomeObject();
  Class<?> c = o.getClass();
  System.out.println(c);                          // class B
  Field[] fields = c.getDeclaredFields();
  Field.setAccessible(fields,true);
  for (Field f : fields) {
    Object x = f.get(o);                          // private int B.number=671
    System.out.println(f+"="+x);                  // private int B.year=2013
  }
```

- `getClass()` used on objects; `.class` used on types (e.g., `int.class`, `List.class`)
- `Class` is **generic**: `A.class` has type `Class<A>`
- `getDeclaredFields` returns all the fields (public and non-public) but **not** the fields declared in the parent class(es)
- `getFields` returns all the **public** fields, **including** the fields declared in the parent class(es)
- `setAccessible` is controlled by the **security manager** of the JVM

```
Field f = c.getDeclaredField("number");    // looks only in B
f.setAccessible(true);
f.set(o, 745); // or  f.set(o, Integer.valueOf(745));
f = c.getDeclaredField("year");
f.setAccessible(true);
f.set(o, 2014);
f = c.getField("course");                  // looks only for public
f.setAccessible(true);
f.set(o, "cs");
System.out.println(o);                     // cs-745(2014)
System.out.println(c.getSuperclass());     // class A
```

- `f.set(o, "cs")` means `o.name = "cs"`,
  where `name` is the name of field `f`
- `getSuperclass()` is `null` for `Object`, interfaces and primitive types
  (primitive types have a corresponding `Class` object)

```
    Class<Reflection> cr = Reflection.class;
    Reflection r = cr.newInstance();
    for (Method m : cr.getMethods()) {
      System.out.println(m);          // dbl + main + methods of Object
      if (m.getName().equals("dbl"))
        System.out.println(m.invoke(r,"CS")); // CSCS
    }
  }
}
```

- `m.invoke(r,"CS")` makes the call `r.dbl("CS")`
- `newInstance()` uses **default constructor**
- use `getConstructors()` and `getDeclaredConstructors()` to access **other constructors** (of type `java.lang.reflect.Constructor`)
- Constructor **exceptions** are handled **differently** in `Class.newInstance` and in `Constructor.newInstance`

Write a method `explore(Object target)` that prints the names and values of **all fields in target**, including non-public and inherited fields.

```
class A {
        public final int x = 43;
        public final String s = "foo";
        public B another;
}
class B extends A {
        private String[] str = new String[10];
        public int a = 15;
        static double x = .5;
}
```

explore(new B())

```
B.str = [Ljava.lang.String;@103c520    // uses JVM internal representation
B.a = 15
A.x = 43
A.s = foo
A.another = null
```

```
public static void explore(Object target) throws Exception{
   Class<?> c = target.getClass();
  do{
     Field[] fields = c.getDeclaredFields();
     Field.setAccessible(fields,true);
     for (Field f : fields) {
        Object x = f.get(target);
        System.out.println(f + "=" + x);
     }
     c = c.getSuperclass();
   } while (c != null);
}
```

explore(new B())

```
private java.lang.String[] B.str=[Ljava.lang.String;@2aa05bc3
public int B.a=15
static double B.x=0.5
public final int A.x=43
public final java.lang.String A.s=foo
public B A.another=null
```

```
public void explore (Object o) throws Exception {
  Class<?> c = o.getClass();
  do {
    String name = c.getName();
    Field[] fields = c.getDeclaredFields();
    Field.setAccessible(fields, true);
    for (Field f : fields)
      if (!Modifier.isStatic(f.getModifiers()))
        System.out.printf("%s.%s = %s%n", name, f.getName(), f.get(o));
    c = c.getSuperclass();
  } while (c != null);
}
```

explore(new B())

```
B.str = [Ljava.lang.String;@103c520
B.a = 15
A.x = 43
A.s = foo
A.another = null
```

Example 2: forName, isAssignableFrom and asSubclass

```
interface I { // known at compile time
  public String getName ();
}
class SomeClass implements I { // not known at compile time
  public SomeClass(String s) { ... }
  ...
}
```

```
Class<?> c = Class.forName("SomeClass");
if (I.class.isAssignableFrom(c)) {
    Constructor<?> cs = c.getConstructor(String.class);
    I i = (I)cs.newInstance("bar");
    System.out.println(i.getName());
}
```

```
Class<?> c = Class.forName("SomeClass");
if (I.class.isAssignableFrom(c)) {
    Class<? extends I> ci = c.asSubclass(I.class);
    Constructor<? extends I> csi = ci.getConstructor(String.class);
    I i = csi.newInstance("bar");
    System.out.println(i.getName());
}
```

Remarks

- `setAccessible` is not required for `public` fields
  (or package-private within the package, or . . .)
- `setAccessible` may be rejected by a **security manager**:
  - running an applet through a browser JVM
  - `java -Djava.security.manager`
- `Class.forName` uses a **class loader** to find classes
  (default `ClassLoader` looks in **CLASSPATH**)

Example

```
ClassLoader loader = new URLClassLoader(jarURLs);
Class<?> c = Class.forName(...,true,loader);
```

In particular, `java.net.URLClassLoader` makes it straightforward to load and use **remote** classes

- most methods from class `Class` may throw a variety of **exceptions**
  (which, in assignments, need to be caught)

Variable Arguments of java.lang.Class.getMethod

```
class C {
 public void main () {...}
 public void main (String s, Object o) {...}
 public void main (int x, int y) {...}
 public static void main (String[] args) {
  Class<C> c = C.class; // or through Class.forName() or Object.getClass()
  Method m;
  m = c.getMethod("main");
  m = c.getMethod("main", String.class, Object.class);
  m = c.getMethod("main", int.class, int.class);
  m = c.getMethod("main", Integer.TYPE, Integer.TYPE);
  m = c.getMethod("main", Integer.class, Integer.class); // NoSuchMethodEx
  m = c.getMethod("main", String[].class);
  m = c.getMethod("main", String[].class.getComponentType(), Object.class);
 }
}
```

Java 1.5 introduced **annotations**,
which can be applied to fields, methods, classes, ...

Examples:

```
@SuppressWarnings("unchecked")
void someMethod (...) { ... }

@Override public String toString () { ... }
```

some annotations are used at **compile-time** then discarded (`@Override`);
others are available at **runtime** and can be accessed through **reflection**

Definition of `@Override`

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Definition of @SuppressWarnings

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

Definition of @charpov.grader.Test

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Test {
  long timeout () default 1000;  // annotation method with default return value
  double val () default 0;
}
```

Examples:

```
@SuppressWarnings({"deprecation", "serial"})
public class SomeClass { ... }

@Test(val=5) void testAllWords () { ... }

@Test(timeout=3000, val=2) void testAddWordsURL () { ... }
```

Accessing Annotations through Reflection

if an annotation uses `RetentionPolicy.RUNTIME`,
it can be accessed at runtime through reflection

```
Method m = ...
Test annotation = m.getAnnotation(Test.class);
if (annotation != null) {
  long time = annotation.timeout();
  ...
}
```

```
Object o = ...
Class<?> c = o.getClass();
Annotation[] annotations = c.getAnnotations();
if (annotations.length > 0) {
  System.out.println("this object comes from an annotated class:");
  for (Annotation a : annotations) {
    System.out.println(a);
  }
}
```

Example: Annotation and Reflection

Define an annotation:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
@interface CanRun {
    long timeout() default 100;
}
```

Example: Annotation and Reflection

Use the annotation to annotate class methods:

```
class AnnotationRunner {
    public void method1() {
        System.out.println("method1");
    }

    @CanRun
    public void method2() {
        System.out.println("method2");
    }

    @CanRun(timeout = 500)
    void method3() {
        System.out.println("method3");
    }

    @CanRun
    public void method4(int x) {
        System.out.println("method4");
    }
}
```

Example: Annotation and Reflection

Use reflection to investigate annotated methods:

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

public class AnnotationTest {
    public static void main(String[] args) {
        Object runner = new AnnotationRunner();
        Method[] methods = runner.getClass().getDeclaredMethods(); // get all methods
        for (Method method : methods) {
            CanRun annos = method.getAnnotation(CanRun.class); // access annotated methods
            if (annos != null) {
                long time = annos.timeout(); // get timeout
                System.out.println(time);
                try {
                    method.invoke(runner);
                } catch (Exception e) {
                    System.out.println("Can not invoke"+method.getName()+": "+e.getMessage());
                }
            }
        }
    }
}
```