Reflection

• Based on https://docs.oracle.com/javase/tutorial/reflect/

• Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.
Uses of reflection

Extensibility Features
An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.

Class Browsers and Visual Development Environments
A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.

Debuggers and Test Tools
Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set of APIs defined on a class, to insure a high level of code coverage in a test suite.
Drawbacks

Performance Overhead
Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance and should be avoided in sections of code which are called frequently in performance-sensitive applications.

Security Restrictions
Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.

Exposure of Internals
Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.
Classes

• For every class, Java instantiates an immutable instance of java.lang.Class.

• Class provides methods to examine the runtime properties of a class

• Class provides the ability to create new classes and objects

• Class, and instances of Class, support and enable reflection
Class objects

• A Class object is an object that represents the information for a class -- it is *not* an instance of the class

  • For class Foo, Foo f = new Foo( ); gets an instance of the class Foo.

  • We want the object *that is* class Foo.

• Class provides methods to examine the runtime properties of class

• Class provides the ability to create new classes and objects

• Class supports and enables reflection

• Therefore, the first goal to using reflection is to access the Class object for a class.
Retrieving Class objects

- If you have a class name, or an instance of a class, i.e., an object:

```java
Class c = "foo".getClass();
Set<String> s = new HashSet<String>();
Class c = s.getClass();
```

- Access the .class field associated with a class:

```java
Class c = boolean.class;
Class c = java.io.PrintStream.class;
Class c = int[][][].class;
```

- `Class.forName( )` with a fully qualified class name

```java
Class c = Class.forName("com.duke.MyLocaleServiceProvider");
Class cDoubleArray = Class.forName("[D");
Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

- `TYPE` field for Primitive wrappers

```java
Class c = Double.TYPE; // Identical to Double.class
Class c = Void.TYPE; // Identical to Void.class
```
Methods that return Classes

• Class c =
  javax.swing.JButton.class.getSuperclass();
  // Returns the super class for the given class.
import java.lang.annotation.Annotation;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.lang.reflect.TypeVariable;
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import static java.lang.System.out;

public class ClassDeclarationSpy {
    public static void main(String... args) {
        try {
            Class<?> c = Class.forName(args[0]);
            out.format("Class:%n  %s%n", c.getCanonicalName());
            out.format("Modifiers:%n  %s%n", Modifier.toString(c.getModifiers()));
            TypeVariable[] tv = c.getTypeParameters();
            if (tv.length != 0) {
                out.format("  ");
                for (TypeVariable t : tv)
                    out.format("%s ", t.getName());
                out.format("%n%n");
            } else {
                out.format("  -- No Type Parameters --%n%n");
            }
            out.format("Implemented Interfaces:%n");
            Type[] intfs = c.getGenericInterfaces();
            if (intfs.length != 0) {
                for (Type intf : intfs)
                    out.format("  %s%n", intf.toString());
                out.format("%n");
            } else {
                out.format("  -- No Implemented Interfaces --%n%n");
            }
        } catch (ClassNotFoundException x) {
            x.printStackTrace();
        }
    }

    private static void printAncestor(Class<?> c, List<Class> l) {
        Class<?> ancestor = c.getSuperclass();
        if (ancestor != null) {
            l.add(ancestor);
            printAncestor(ancestor, l);
        }
    }
}

out.format("Inheritance Path:%n");
List<Class> l = new ArrayList<Class>();
printAncestor(c, l);
if (l.size() != 0) {
    for (Class<?> cl : l)
        out.format("  %s%n", cl.getCanonicalName());
    out.format("%n");
} else {
    out.format("  -- No Super Classes --%n%n");
}

out.format("Annotations:%n");
Annotation[] ann = c.getAnnotations();
if (ann.length != 0) {
    for (Annotation a : ann)
        out.format("  %s%n", a.toString());
    out.format("%n");
} else {
    out.format("  -- No Annotations --%n%n");
}
public class ClassDeclarationSpy {
    public static void main(String args[]) {
        try {
            Class<?> c = Class.forName(args[0]);
            out.format("Class:%n  %s%n%n", c.getCanonicalName);
            out.format("Modifiers:%n  %s%n%n", Modifier.toString(c.getModifiers()), Modifier.toString(c.getModifiers()));
        }
    }
}

Output:

$ java ClassDeclarationSpy "[Ljava.lang.String;"
Class:
    java.lang.String[]
Modifiers:
    public abstract final
Examining class properties

```java
out.format("Type Parameters:%n");
TypeVariable[] tv = c.getTypeParameters();
if (tv.length != 0) {
    out.format(" ");
    for (TypeVariable t : tv)
        out.format("%s ", t.getName());
    out.format("%n%n");
} else {
    out.format(" -- No Type Parameters --%n%n");
}

out.format("Implemented Interfaces:%n");
Type[] intfs = c.getGenericInterfaces();
if (intfs.length != 0) {
    for (Type intf : intfs)
        out.format(" %s%n", intf.toString());
    out.format("%n");
} else {
    out.format(" -- No Implemented Interfaces --%n%n");
}

Output:
Type Parameters:
   -- No Type Parameters --

Implemented Interfaces:
   interface java.lang.Cloneable
   interface java.io.Serializable
```
### Getting class members

#### Fields

<table>
<thead>
<tr>
<th>Class API</th>
<th>members?</th>
<th>inherited?</th>
<th>private?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDeclaredField(String)</code></td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><code>getField(String)</code></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td><code>getDeclaredFields()</code></td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><code>getFields()</code></td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
## Getting class members

### Methods

<table>
<thead>
<tr>
<th>Class API</th>
<th>members?</th>
<th>inherited?</th>
<th>private?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDeclaredMethod(String)</code></td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><code>getMethod(String)</code></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td><code>getDeclaredMethods()</code></td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><code>getMethods()</code></td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
You can use these to call methods, even private methods

```java
Method method =
    object.getClass().getDeclaredMethod(methodName);
method.setAccessible(true);
Object r = method.invoke(object);
```

This is not common, but some programs do access methods. For testing frameworks, e.g., you can use *annotations* to mark functions to be tested, and execute them.
## Getting class members

### Constructors

<table>
<thead>
<tr>
<th>Class API</th>
<th>members?</th>
<th>inherited?</th>
<th>private?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDeclaredConstructor()</code></td>
<td>NO</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td><code>getConstructor()</code></td>
<td>NO</td>
<td>N/A</td>
<td>NO</td>
</tr>
<tr>
<td><code>getDeclaredConstructors()</code></td>
<td>YES</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td><code>getConstructors()</code></td>
<td>YES</td>
<td>N/A</td>
<td>NO</td>
</tr>
</tbody>
</table>
Other methods exist

• For analyzing method calls
• For creating arrays and instances of objects
• for changing the values of fields, including private fields
• You can essentially find out anything you want about classes, methods and fields with reflection, and change values of anything
Nested classes

From
https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
Nested classes

- Nested classes can be static or non-static
- Static classes do not have access to members of the enclosing class (other than themselves, of course)
- Non-static nested classes are called *inner classes*
- static nested classes are called *static nested classes*.
Why nested classes?

It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.
Accessing and creating static nested classes

Static nested classes are accessed using the enclosing class name:
OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.
Accessing and creating inner classes

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```java
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance of the outer class, it cannot define any static members itself.
Local and anonymous classes

There are two additional types of inner classes. You can declare an inner class within the body of a method. These classes are known as local classes. You can also declare an inner class within the body of a method without naming the class. These classes are known as anonymous classes.
Within the phone number class, only final or effectively final (Java 8) fields of the outer class and scope can be referenced. Effectively final fields are not changed after initialization.

Values of the fields are captured at class creation time.

Other fields can be accessed by passing them in as function arguments.
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello "+ name);
            }
        }
    }
}
HelloWorld englishGreeting = new EnglishGreeting();

HelloWorld frenchGreeting = new HelloWorld() {
    String name = "tout le monde";
    public void greet() {
        greetSomeone("tout le monde");
    }
    public void greetSomeone(String someone) {
        name = someone;
        System.out.println("Salut " + name);
    }
};

englishGreeting.greet();
frenchGreeting.greetSomeone("Fred");
spanishGreeting.greet();
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }
    }
}
anonymous class example

```java
HelloWorld englishGreeting = new EnglishGreeting();

HelloWorld frenchGreeting = new HelloWorld() {
    String name = "tout le monde";
    public void greet() {
        greetSomeone("tout le monde");
    }
    public void greetSomeone(String someone) {
        name = someone;
        System.out.println("Salut " + name);
    }
};

...englishGreeting.greet();
frenchGreeting.greetSomeone("Fred");
spanishGreeting.greet();
...}
```

Instantiating an EnglishGreeting object

An anonymous class. The constructor is an Interface name (HelloWorld), the anonymous class implements the interface.

Code in SayHello that Invokes a method within the anonymous class
Lambda expressions

Lambda expressions are a form of anonymous class that implement interfaces with only one method.

```java
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }
    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
                myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
                myApp.operateBinary(20, 10, subtraction));
    }
}
```
When to use these

- **Local class**: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).

- **Anonymous class**: Use it if you need to declare fields or additional methods, but only one instance.

- **Lambda expression**:
  - Use it if you are encapsulating a single unit of behavior that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
  - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).

- **Nested class**: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to method local variables or method parameters.
  - Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.