Reflection

Agenda

- Introduction
- Reflection API
- Class, Constructors & Methods Access
- Method Invocation
- Method & Fields Accessibilities

Introduction - Reflection

- **Reflection** is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.
- Java Package: java.lang.reflect
- Gives us information about the class to which an object belongs and also the methods of that class.
- With reflection we can invoke methods at runtime irrespective of the access specifier used with them.
- With Reflection, code is easier to **maintain** & **extend**.

Reflection API

Reflection

Reflection gives us information about:

Class The getClass() method

Constructors The getConstructors() method

Methods The getMethods()

Example 1. Class Information

```
2  public class testReflection {
3
4      public static void main(String[] args) {
5          MyClass myClass = new MyClass();
6          Class clazz = myClass.getClass();
7          System.out.println(clazz.getName());
8      }
9  }
```

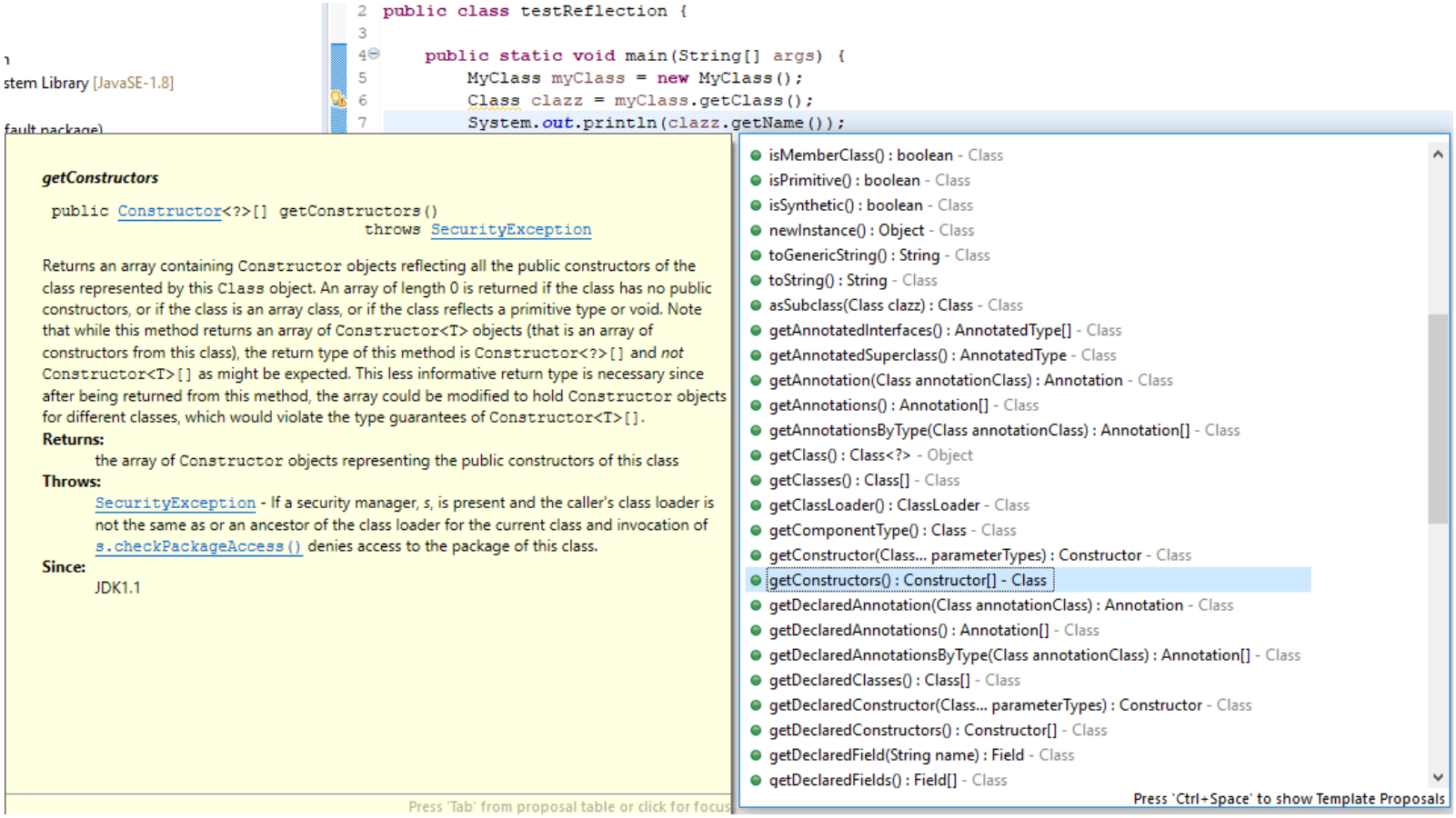
```
2  public class MyClass {
3      private int num = 0;
4
5      public MyClass() {
6          this.num=5;
7      }
8  }
```

Output:
MyClass

Class Exercise

Run the previous program & check with debug the output.

Debug


stem Library [JavaSE-1.8]
fault package)
2 public class testReflection {
3
4 public static void main(String[] args) {
5 MyClass myClass = new MyClass();
6 Class clazz = myClass.getClass();
7 System.out.println(clazz.getName());
getConstructors
public Constructor<?>[] getConstructors()
throws SecurityException
Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object. An array of length 0 is returned if the class has no public constructors, or if the class is an array class, or if the class reflects a primitive type or void. Note that while this method returns an array of Constructor<T> objects (that is an array of constructors from this class), the return type of this method is Constructor<?>[] and not Constructor<T>[] as might be expected. This less informative return type is necessary since after being returned from this method, the array could be modified to hold Constructor objects for different classes, which would violate the type guarantees of Constructor<T>[].
Returns:
the array of Constructor objects representing the public constructors of this class
Throws:
SecurityException - If a security manager, s, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of s.checkPackageAccess() denies access to the package of this class.
Since:
JDK1.1
Press 'Tab' from proposal table or click for focus
isMemberClass() : boolean - Class
isPrimitive() : boolean - Class
isSynthetic() : boolean - Class
newInstance() : Object - Class
toGenericString() : String - Class
toString() : String - Class
asSubclass(Class clazz) : Class - Class
getAnnotatedInterfaces() : AnnotatedType[] - Class
getAnnotatedSuperclass() : AnnotatedType - Class
getAnnotation(Class annotationClass) : Annotation - Class
getAnnotations() : Annotation[] - Class
getAnnotationsByType(Class annotationClass) : Annotation[] - Class
getClass() : Class<?> - Object
getClasses() : Class[] - Class
getClassLoader() : ClassLoader - Class
getComponentType() : Class - Class
getConstructor(Class... parameterTypes) : Constructor - Class
getConstructors() : Constructor[] - Class
getDeclaredAnnotation(Class annotationClass) : Annotation - Class
getDeclaredAnnotations() : Annotation[] - Class
getDeclaredAnnotationsByType(Class annotationClass) : Annotation[] - Class
getDeclaredClasses() : Class[] - Class
getDeclaredConstructor(Class... parameterTypes) : Constructor - Class
getDeclaredConstructors() : Constructor[] - Class
getDeclaredField(String name) : Field - Class
getDeclaredFields() : Field[] - Class
Press 'Ctrl+Space' to show Template Proposals

Example 2. Constructor Access

```
2  public class MyClass {
3      private int num = 0;
4
5      public MyClass() {
6          this.num=5;
7      }
8
9      public MyClass(int num) {
10         this.num=num;
11     }
12
13     public MyClass(int num, int p) {
14         this.num=num;
15     }
16
17     public void setNum(int num) {
18         this.num = num;
19     }
20 }
```

```
2  public class testReflection {
3
4      public static void main(String[] args) {
5          MyClass myClass = new MyClass();
6          Class clazz = myClass.getClass();
7          System.out.println(clazz.getName());
8
9          for (int i=0; i<clazz.getConstructors().length; i++) {
10             System.out.println("Consructor " + i + ": " +
11                     clazz.getConstructors()[i].getName() +
12                     " Num of params: " + clazz.getConstructors()[i].getParameterCount());
13         }
14     }
15 }
```

MyClass
Consructor 0: MyClass Num of params: 0
Consructor 1: MyClass Num of params: 2
Consructor 2: MyClass Num of params: 1

Class Exercise

Test the example with Constructors.


MyClass.java


testReflection.java

Method Invocation

We can invoke an method with reflection if we know its name and parameter types.

We use below two methods for this purpose **getDeclaredMethod() & Invoke()**

Class.**getDeclaredMethod**(name, parametertype)

..

Method.**invoke**(Object, parameter)

Method Invocation With Reflection

```java
public class MyClass {
    private int num = 0;

    public MyClass() {
        this.num=5;
    }

    public MyClass(int num) {
        this.num=num;
    }

    public MyClass(int num, int p) {
        this.num=num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public void printMe(String str) {
        System.out.println("printMe method was called: " + str);
    }
}
```

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

public class testReflection {

    public static void main(String[] args) {
        MyClass myClass = new MyClass();
        Class clazz = myClass.getClass();
        System.out.println(clazz.getName());

        try {
            Method myMethod = clazz.getDeclaredMethod("printMe", String.class);
            myMethod.invoke(myClass, "myStr");
        } catch (NoSuchMethodException | SecurityException | IllegalAccessException |
                 IllegalArgumentException | InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

MyClass
printMe method was called: myStr

Class Exercise

Run the Method Invocation code in your IDE & check the result with debug.

Private Methods & Variables Access

With reflection we can **access the private variables and methods**.

We use below two methods for this purpose:

Class.getDeclaredField(FieldName)

..
Field.setAccessible(true)

Change Method Accessibility

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

public class testReflection {

    public static void main(String[] args) {
        MyClass myClass = new MyClass();
        Class clazz = myClass.getClass();
        System.out.println(clazz.getName());

        try {
            Method myMethod = clazz.getDeclaredMethod("printMe", String.class);
            myMethod.setAccessible(true);
            myMethod.invoke(myClass, "myStr");
        } catch (NoSuchMethodException | SecurityException | IllegalAccessException |
                 IllegalArgumentException | InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

```java

public class MyClass {
    private int num = 0;

    public MyClass() {
        this.num=5;
    }

    private void printMe(String str) {
        System.out.println("printMe method was called: " + str);
    }
}
```

MyClass
printMe method was called: myStr

Class Exercise

Run the Accessibility example and observe the result in debug.

Private Field Accessibility

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

public class testReflection {

    public static void main(String[] args) {
        MyClass myClass = new MyClass();
        Class clazz = myClass.getClass();
        System.out.println(clazz.getName());

        try {
            Method myMethod = clazz.getDeclaredMethod("printMe", String.class);
            myMethod.setAccessible(true);
            myMethod.invoke(myClass, "myStr");

            Field field = clazz.getDeclaredField("num");
            field.setAccessible(true);
            System.out.println("private field name:" + field.getName() + " Value:" + field.get(myClass));
        } catch (NoSuchMethodException | SecurityException | IllegalAccessException |
                 IllegalArgumentException | InvocationTargetException | NoSuchFieldException e) {
            e.printStackTrace();
        }
    }
}
```

```java
public class MyClass {
    private int num = 0;

    public MyClass() {
        this.num=5;
    }

    private void printMe(String str) {
        System.out.println("printMe method was called: " + str);
    }
}
```

MyClass
printMe method was called: myStr
private field name:num Value: 5

Class Exercise

Run the Field Accessibility example and observe the result in debug.

Reflection Disadvantages

- **Performance Overhead:** Reflective operations have slower performance than their non-reflective code. We should avoid reflection code which is called frequently.

- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior.

Thank You!