Table of Contents

1 Introduction

Process of analyzing all the capabilities of Class at run time can be achieved through Reflection API.
To know all the details about variable, methods, classes, etc.

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

Where we use reflection?

It may not be very much useful in project development.
It may be very much useful in Product development.

Ex 1 –

Most of the compiler use Reflection API to read all the declaration information of Class

```
Private Class A{ // Compiler raise an error.
        ---
        ---
}
```

Compiler will raise an error saying "private" is not allowed. How does compiler raise an error?
n Compiler has set of rules; it has rules for access modifier rules.
It reads Class declaration modifier by using Reflection APL and generates an error.

Ex 2 –

JVM uses Reflection to read all the declaration information of a Class, to execute set of rules of JVM.

```
Class A{
        Private A(int i){
        }
}
Class Test{
        Public static void main(String[] args){
                Class c = Class.forName("A");
                A a = (A) c.newInstance()// JVM throw an error by saying instantiation.
        }
}
```

Ex-

Most of the tools use Reflection API used to inspect class level information at without referring Actual class names.

2 Java.lang.reflect Package

Java.lang.Class – To get complete information about a Class

Java.lang.reflect.Field – To get complete information about a Field in a java class

Java.lang.reflect.Method – To get complete declaration information about a java method

Java.lang.reflect.Constructor – To get complete declaration information about a java constructor

Java.lang.reflect.Modifier – To get complete access modifier information about a java members

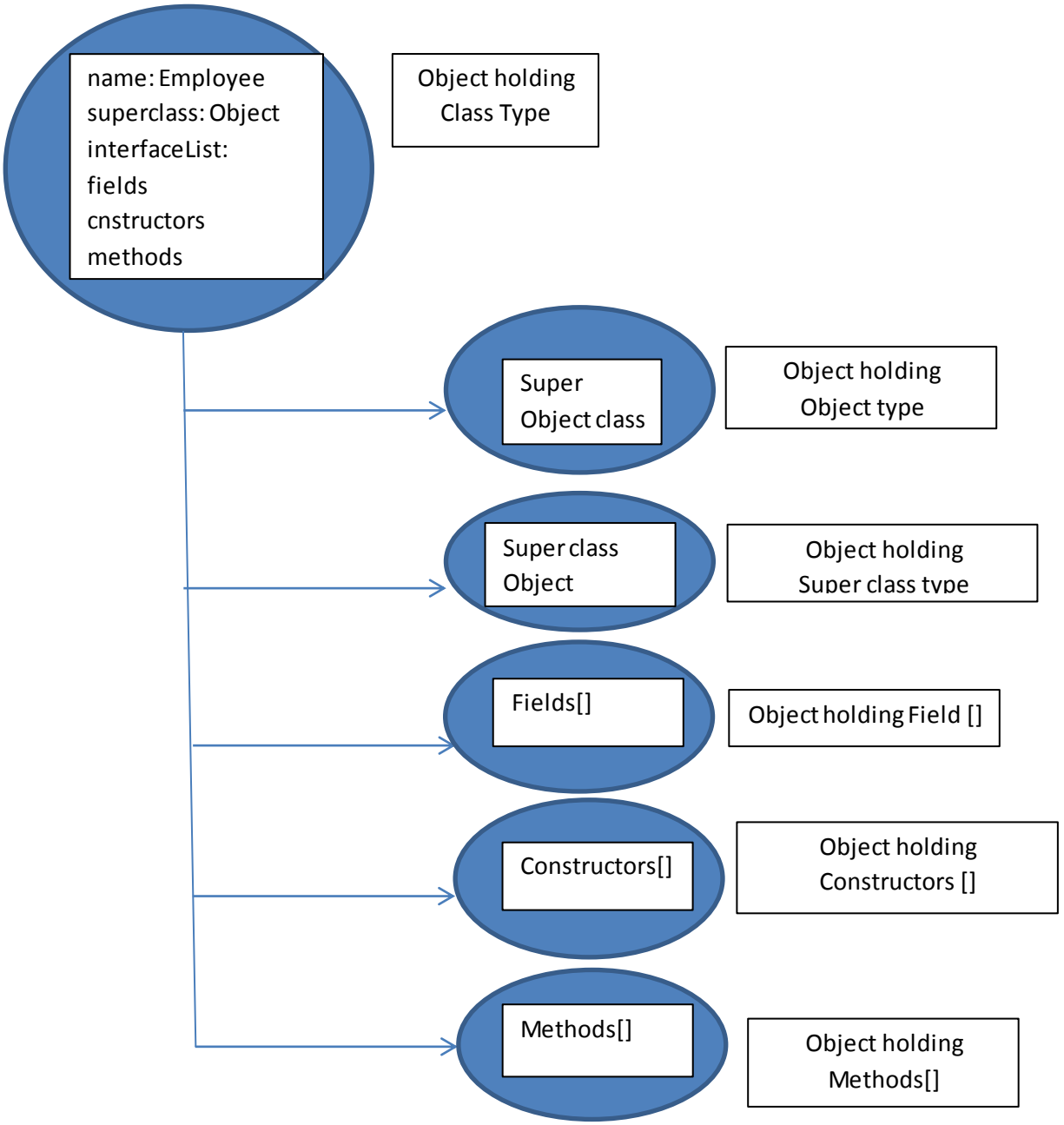
Ex –

```
Public Class Employee implements Serializable, Cloneable{
        Public int eno;
        Public String ename;
        Public String eaddr;
        Public Employee (){
        }
        Public Employee (int I, String ename, String addr){
        }
        Public Employee (){
        }
        Public void add () {
        }
}
```

Most of the inormatoin related to a class such as fields, methods, constructors, etc are usually managed in the form of individual arrays of Object/Objects like below

Class Object – If manage all the class level information
Super class Object – to manage Super class object (Ex Object class metadata)
Interfaces – array of Class objects are used to manage all the implemented interfaces.
Fields – Array of Field objects used to manage all the fields information
Constructors - array of constructor objects used to manage all the constructors information.
Methods – arrays of methods objects used to manage all the methods information in a class.

Class object holds all the above different objects, so if you get hold of Class object then you can retrieve all the other members' information of a class by using reflection.



Object Graph of Class Type Object

3 Reflection - Class

Before you can do any inspection on a class you need to obtain its java.lang.Class object. All types in Java including the primitive types (int, long, float etc.) including arrays have an associated Class object.

There are 3 ways to create **Class object**.

1. If you don't know the name at compile time, but have the class name as a string at runtime, you can do like this:

   ```
   String className = ... //obtain class name as string at runtime
   Class class = Class.forName(className);
   ```

   When using the Class.forName() method you must supply the fully qualified class name. That is the class name including all package names.

   Class.forName() will raise ClassNotFoundException exception if Class file is not available.

   If .class if available then JVM will collect all the declarative metadata of a class and will be stored in "Class Object"

2. MyObject obj = ne MyObject ();`
   Class c = obj.getClsss();
   Whenever Employes class is loaded "Class" object is created.

3. If you know the name of the class at compile time you can obtain a Class object like this:
   Class myObjectClass = MyObject.class
   When class is compiled class property will be added to an Object).
   Whenever we compile our class class property is added to object.

3.1 Get Class Information

1. public String getName() – To get fully qualified name of a classname
2. public String getSimpleName()- to get just only class name
3. public Class getSuperClass() – To get super class
4. public Class[] getInterface() – get list of interfaces.
5. public int getModifiers() – To get the modifiers
   public static String Modifiers.toString(int i)
6. Package getPackage() – To obtain information about the package from a Class
7. Annotation[] getAnnotations() – to get the information about annotations.

Ex -

```
public Class Employee implements Serializable, Cloneable{
}
public Class Test  throws Exception {
        Public static void main(String[] args){
                Class c = Class.forName("Employee");
                System.out.println("Name" + c.getName());
                System.out.println("Super class" + c.getSuperClass().getName());
                Class[] cl = c.getInterfaces());
                For (Class c1: cl){
                        System.out.println("Interface name" + c.getName());
                }
                Int i= c.getModifiers()
                System.out.println("Class modifiers" + Modifer.toString(i));

        }
}
```

4 Reflection - Fields

Using Java Reflection you can inspect the fields (member variables) of classes and get / set them at runtime. This is done via the Java class java.lang.reflect.Field.

4.1 Obtaining Field Objects

Use below methods to retrieving fields from a class.

Obtaining/Accessing public fields -

1. Fields[] fields = c.getFileds();
   This method is used to get all the **public fields from both super and current class.**
2. Fields fields = c.getField("someField")
   This method can be used when you know the exact name of a field in the class

Obtaining/Accessing private/public fields -

3. Fields[] fields = c.getDeclaredFields();
   This method is used to get all the fields from current class.
4. Fields field = c.getDeclaredField("somePrivateField");
   In reflection once you retrieve the private field you should turn off the accessible check on compiler by invoking following method –
   privateStringField.setAccessible(true);

4.2 Retrieving metadata of a field

Use below methods to retrieving metadata of a field.

1. Sting String getName(); - To get name of the field
2. Public Class getType(); - To get type of the field
3. Public XXX get(Field ref) - To get value of the field
4. Public void set(objetInstance, value) - To set value of the field
5. public int getModifiers() – To get the modifiers
   public static String Modifiers.toString(int i)

Ex -

```
public Class Employee {
        Public static int eno=111;
        static String ename="test";
        static String eaddr="Hyd";
}
public Class Test  throws Exception {
        Public static void main(String[] args){
                Employee e =  new Employee();
```

```
                Class c = e.getClass();
                Fields[] fields = c.getDeclaredFields());
                For (Fields field: fields){
                        System.out.println("field name" + field.getName());
                        Class c = field.getType();
                        System.out.println("field name" + c.getName());
                        Int i= field.getModifiers()
                        System.out.println("Field modifiers" + Modifer.toString(i));
                        System.out.println("field value" + field.get(feild));
                }
        }
}
```

5 Reflection - Methods

Using Java Reflection you can inspect the methods of classes and invoke them at runtime. This is done via the Java class java.lang.reflect.Method

5.1 Obtaining Method Objects

Use below methods to retrieving methods information from a class.

Obtaining/Accessing public methods-

1. Method [] methods = c.getMethods();
   This method is used to get all the **public methods from both super and current class.**
2. Method method = c.getMethods("doSomething", new Class[]{String.class});
   If you know the precise parameter types of the method you want to access, then you can use above method to obtain Method object directly.
   This example returns the public method named "doSomething", in the given class which takes a String as parameter.
   - If no method matches the given method name and arguments, in this case String.class, aNoSuchMethodException is thrown.
   - If the method you are trying to access takes no parameters, pass null as the parameter type array, like this:
     Method method = aClass.getMethod("doSomething", null);

Obtaining/Accessing public/private methods-

To access a private method you will need to call the Class.getDeclaredMethod(String name, Class[] parameterTypes) or Class.getDeclaredMethods() method.

3. Method [] methods = c.getDeclaredMethods();
   This method is used to get **all the methods from current class.**
4. Method [] methods = c.getDeclaredMethods("getPrivateString", null);

To turn off the access checks for a particular Method instance, use below method for reflection only.

privateStringMethod.setAccessible(true);

5.2 Invoking methods using Method Object

You can invoke a method like this: get method that takes a String as argument –

```
Method method = MyObject.class.getMethod("doSomething", String.class);
Object returnValue = method.invoke(null, "parameter-value1");
```

- The null parameter is the object you want to invoke the method on. **If the method is static you supply null instead of an object instance**.
- If invoking method (doSomething(String.class)) is a not static method then you n**eed to supply a valid MyObject instance instead of null**.

- The Method.invoke(Object target, Object ... parameters) method takes an optional amount of parameters, but you must supply exactly one parameter per argument in the method you are invoking. In this case it was a method taking a String, so one String must be supplied.

5.3 Retrieving methods information

Use below methods to retrieving metadata of a method.

1. Sting String getName(); - To get name of the method
2. Public Class getRetunType(); - To get return type of the method
3. public int getModifiers() – To get the modifiers
   public static String Modifiers.toString(int i)
4. Public Class[] getParametersTypes(); - To get parameter types of the method
5. Public Class[] getExceptionTypes(); - To get exception types of the method

Ex –

```
Public Class Employee {
        Public void add (int eno, String name, Strng addr) throws ClassNotFoundException {}
        Public void search(int eno) throws ArithmeticException, InterruptedException{}
        Public void delete(int eno) throws SQLException{ }
}
public Class Test  throws Exception {
        Public static void main(String[] args){
                Class c = Employee.class;
                Method [] methods= c.getDeclaredMethods());
                For (Method method: methods){
                        System.out.println("method name" + method.getName());
                        Class c = method. getRetunType();
                        System.out.println("Method retruntype + c.getName());
                        Int i= method.getModifiers()
                        System.out.println("Class modifiers" + Modifer.toString(i));
                        Class[] cls = c.getParameterTypes());
                        For (Class c1: cls){
                                System.out.println("Parameter name" + c1.getName());
                        }
                        Class[] clsExcps = c.getExceptionTypes());
                        For (Class c2: clsExcps){
                                System.out.println("Parameter name" + c2.getName());
                        }
                }
        }
}
```

5.4 Retrieving Getters and Setters methods.

Using Java Reflection you can inspect the methods of classes and invoke them at runtime. This can be used to detect what getters and setters a given class has. You cannot ask for getters and setters explicitly, so you will have to scan through all the methods of a class and check if each method is a getter or setter.

First let's establish the rules that characterizes getters and setters:

- Getter
  A getter method have its name start with "get", take 0 parameters, and returns a value.
- Setter
  A setter method have its name start with "set", and takes 1 parameter.

  Setters may or may not return a value. Some setters return void, some the value set, others the object the setter were called on for use in method chaining. Therefore you should make no assumptions about the return type of a setter.

Ex –

```
public static void printGettersSetters(Class aClass){
        Method[] methods = aClass.getMethods();

        for(Method method : methods){
          if(isGetter(method)) System.out.println("getter: " + method);
          if(isSetter(method)) System.out.println("setter: " + method);
        }
}

public static boolean isGetter(Method method){
        if(!method.getName().startsWith("get"))     return false;
        if(method.getParameterTypes().length != 0)   return false;
        if(void.class.equals(method.getReturnType()) return false;
        return true;
}

public static boolean isSetter(Method method){
        if(!method.getName().startsWith("set")) return false;
        if(method.getParameterTypes().length != 1) return false;
        return true;
}
```

6 Reflection - Constructors

Using Java Reflection you can inspect the constructors of classes and instantiate objects at runtime. This is done via the Java class java.lang.reflect.Constructor. This text will get into more detail about the JavaConstructor object.

6.1 Obtaining Constructor Objects

- Use below methods to retrieving constructor information from a class.
  1. Constructor [] cons = c.getConstructors();
     This method is used to get all the **public constructors from current class. We can't get super class constructors.**
  2. Constructor [] cons = c.getDeclaredConstructors ();
     This method is used to get **all the constructors from current class.**

- If you know the precise parameter types of the constructor you want to access, you can do so rather than obtain the array all constructors. This example returns the public constructor of the given class which takes a Stringas parameter:

```
Class aClass = ...//obtain class object
Constructor constructor =  aClass.getConstructor(new Class[]{String.class});
```

  If no constructor matches the given constructor arguments, in this case String.class, aNoSuchMethodException is thrown.

6.2 Instantiating Objects using Constructor Object

You can instantiate an object like this: get constructor that takes a String as argument

```
Constructor constructor = MyObject.class.getConstructor(String.class);
MyObject myObject = (MyObject) constructor.newInstance("constructor-arg1");
```

The Constructor.newInstance() method takes an optional amount of parameters, but you must supply exactly one parameter per argument in the constructor you are invoking. In this case it was a constructor taking a String, so one String must be supplied.

6.3 Retrieving metadata of a constructor

Use below methods to retrieving metadata of a constructor.

1. Sting String getName(); - To get name of the constructor
2. public int getModifiers() – To get the modifiers
   public static String Modifiers.toString(int i)
3. Public Class[] getParametersTypes(); - To get parameter types of the constructor
4. Public Class[] getExceptionTypes(); - To get exception types of the constructor

Ex –

```
Public Class Employee {
        Public Employee (int eno, String name, Strng addr) throws ClassNotFoundException{}
        Public Employee(int eno) throws ArithmeticException, InterruptedException{}
        Public Employee () throws SQLException{ }
}
public Class Test  throws Exception {
        Public static void main(String[] args){
                Class c = Employee.class;
                Method [] methods= c.getDeclaredConstructors());
                For (Constructor con: constructors){
                        System.out.println("constructor ame" + con.getName());
                        Int i= con.getModifiers()
                        System.out.println("Constructor modifiers" + Modifer.toString(i));
                        Class[] cls = con.getParameterTypes());
                        For (Class c1: cls){
                                System.out.println("Parameter name" + c1.getName());
                        }
                        Class[] clsExcps = con.getExceptionTypes());
                        For (Class c2: clsExcps){
                                System.out.println("Parameter name" + c2.getName());
                        }
                }
        }

}
```

7 Reflection - Annotations

Using Java Reflection you can access the annotations attached to Java classes at runtime.

Sample Annotation declaration –

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyAnnotation {
    public String name();
    public String value();
}
```

Sample Annotation usage –

```
@MyAnnotation(name="someName",  value = "Hello World")
public class TheClass {
}
```

- @Retention(RetentionPolicy.RUNTIME) means that the annotation can be accessed via reflection at runtime. If you do not set this directive, the annotation will not be preserved at runtime, and thus not available via reflection.
- @Target(ElementType.TYPE) means that the annotation can only be used ontop of types (classes and interfaces typically). You can also specify METHOD or FIELD, or you can leave the target out alltogether so the annotation can be used for both classes, methods and fields

7.1 Retrieving metadata of Annotations

You can retrieve the annotations of a class, method or field at runtime. He re is an example that accesses the class annotations:

Once you got the annotation object, you can invoke getXXX() annotation object to retrieve annotation values programmatically.

7.1.1 Retrieve Class Annotations

You can access the annotations of a class, method or field at runtime. Here is an example that accesses the class annotations:

```
Class aClass = TheClass.class;
Annotation[] annotations = aClass.getAnnotations();
for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
}
```

You can also access a specific class annotation like this:

```
Class aClass = TheClass.class;
Annotation annotation = aClass.getAnnotation(MyAnnotation.class);
if(annotation instanceof MyAnnotation){
    MyAnnotation myAnnotation = (MyAnnotation) annotation;
    System.out.println("name: " + myAnnotation.name());
    System.out.println("value: " + myAnnotation.value());
}
```

7.1.2 Retrieve Method Annotations

Here is an example of a method with annotations:

```
public class TheClass {
    @MyAnnotation(name="someName",  value = "Hello World")
    public void doSomething(){}
}
```

You can access method annotations like this:

```
Method method = ... //obtain method object
Annotation[] annotations = method.getDeclaredAnnotations();
for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
}
```

You can also access a specific method annotation like this:

```
Method method = ... // obtain method object
Annotation annotation = method.getAnnotation(MyAnnotation.class);
if(annotation instanceof MyAnnotation){
    MyAnnotation myAnnotation = (MyAnnotation) annotation;
    System.out.println("name: " + myAnnotation.name());
    System.out.println("value: " + myAnnotation.value());
}
```

7.1.3 Retrieve Parameter Annotations

It is possible to add annotations to method parameter declarations too. Here is how that looks:

```
public class TheClass {
    public static void doSomethingElse(
```

```
            @MyAnnotation(name="aName", value="aValue") String parameter){
        }
    }
```

You can access parameter annotations from the Method object like this:

```
    Method method = ... //obtain method object
    Annotation[][] parameterAnnotations = method.getParameterAnnotations();
    Class[] parameterTypes = method.getParameterTypes();
    int i=0;
    for(Annotation[] annotations : parameterAnnotations){
        Class parameterType = parameterTypes[i++];
        for(Annotation annotation : annotations){
            if(annotation instanceof MyAnnotation){
                MyAnnotation myAnnotation = (MyAnnotation) annotation;
                System.out.println("param: " + parameterType.getName());
                System.out.println("name : " + myAnnotation.name());
                System.out.println("value: " + myAnnotation.value());
            }
        }
    }
```

Notice how the Method.getParameterAnnotations() method returns a two-dimensional Annotation array, containing an array of annotations for each method parameter.

7.1.4 Retrieve Field Annotations

Here is an example of a field with annotations:

```
    public class TheClass {
        @MyAnnotation(name="someName",  value = "Hello World")
        public String myField = null;
    }
```

You can access field annotations like this:

```
    Field field = ... //obtain field object
    Annotation[] annotations = field.getDeclaredAnnotations();
    for(Annotation annotation : annotations){
        if(annotation instanceof MyAnnotation){
            MyAnnotation myAnnotation = (MyAnnotation) annotation;
            System.out.println("name: " + myAnnotation.name());
            System.out.println("value: " + myAnnotation.value());
        }
    }
```

You can also access a specific field annotation like this:

```
Field field = ... // obtain method object
Annotation annotation = field.getAnnotation(MyAnnotation.class);
if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
}
```

8 Reflection - Generics

In most of the articles and forums its mentioned that all **Java Generics information is erased at compile** time so that you cannot access any of that information at runtime. **This is not entirely true though.**
It is possible to access generics information at runtime in a handful of cases.

8.1 The Generics Reflection Rule of Thumb

Using Java Generics typically falls into one of two different situations:

- Declaring a class/interface as being parameterizable.
- Using a parameterizable class.

Declaring a class/interface as being parameterizable.

- When you write a class or interface you can specify that it should be paramerizable. This is the case with the java.util.List interface. Rather than create a list of Object you can parameterize java.util.List to create a list of say String.
- When runtime inspecting a parameterizable type itself, like java.util.List, there is no way of knowing what type is has been parameterized to. This makes sense since the type can be parameterized to all kinds of types in the same application.
- you cannot see on a type itself what type it is parameterized to a runtime, So we can't get Generic information of declarations.

Using a parameterizable class.

- But, when you inspect the method or field that declares the use of a parameterized type, you can see at runtime what type the paramerizable type was parameterized to. In short:
- You can see it in fields and methods where it is used and parameterized. Its concrete parameterizations in other words.

8.2 Generic Method Return Types

If you have obtained a java.lang.reflect.Method object it is possible to obtain information about its generic return type. This cannot be any of the Method objects in the parameterized type, but in the class that uses the parameterized type.
Ex -

```
public class MyClass {
        protected List<String> stringList = ...;
        public List<String> getStringList(){
                return this.stringList;
        }
}
```

In this class it is possible to obtain the generic return type of the getStringList( ) method. In other words, it is possible to detect that getStringList() returns a List<String> and not just a List. Here is how:

```
Method method = MyClass.class.getMethod("getStringList", null);
Type returnType = method.getGenericReturnType();
if(returnType instanceof ParameterizedType){
        ParameterizedType type = (ParameterizedType) returnType;
        Type[] typeArguments = type.getActualTypeArguments();
        for(Type typeArgument : typeArguments){
                Class typeArgClass = (Class) typeArgument;
                System.out.println("typeArgClass = " + typeArgClass);
        }
}
```

This piece of code will print out the text "typeArgClass = java.lang.String". The Type[] array typeArguments array will contain one item - a Class instance representing the class java.lang.String. Class implements the Type interface.

8.3 Generic Method Parameter Types

You can also access the generic types of parameter types at runtime via Java Reflection.
Here is an example class with a method taking a parameterized List as parameter:

```
public class MyClass {
        protected List<String> stringList = ...;
        public void setStringList(List<String> list){
                this.stringList = list;
        }
}
```

You can access the generic parameter types of the method parameters like this:

```
Type[] genericParameterTypes = method.getGenericParameterTypes();
for(Type genericParameterType : genericParameterTypes){
    if(genericParameterType instanceof ParameterizedType){
        ParameterizedType aType = (ParameterizedType) genericParameterType;
        Type[] parameterArgTypes = aType.getActualTypeArguments();
        for(Type parameterArgType : parameterArgTypes){
            Class parameterArgClass = (Class) parameterArgType;
            System.out.println("parameterArgClass = " + parameterArgClass);
    }
  }
}
```

This code will print out the text "parameterArgType = java.lang.String". The Type[] array parameterArgTypes array will contain one item - a Class instance representing the class java.lang.String. Class implements the Type interface.

8.4 Generic Field Types

It is also possible to access the generic types of public fields. Fields are class member variables - either static or instance variables.

Here is the example from earlier, with an instance field called stringList.

```
public class MyClass {
    public List<String> stringList = ...;
}
Field field = MyClass.class.getField("stringList");
Type genericFieldType = field.getGenericType();
if(genericFieldType instanceof ParameterizedType){
    ParameterizedType aType = (ParameterizedType) genericFieldType;
    Type[] fieldArgTypes = aType.getActualTypeArguments();
    for(Type fieldArgType : fieldArgTypes){
        Class fieldArgClass = (Class) fieldArgType;
        System.out.println("fieldArgClass = " + fieldArgClass);
    }
}
```

This code will print out the text "fieldArgClass = java.lang.String". The Type[] array fieldArgTypes array will contain one item - a Class instance representing the class java.lang.String. Class implements the Type interface.

9 Reflection – Arrays

Working with arrays in Java Reflection can be a bit tricky at times. Especially if you need to obtain the Class object for a certain type of array, like int[] etc.

9.1 java.lang.reflect.Array

Working with arrays via Java Reflection is done using the java.lang.reflect.Array class.

Do not confuse this class with the java.util.Arrays class in the Java Collections suite, which contains utility methods for sorting arrays, converting them to collections etc.

9.2 Creating Arrays

Creating arrays via Java Reflection is done using the java.lang.reflect.Array class. Here is an example showing how to create an array:

```
int[] intArray = (int[]) Array.newInstance(int.class, 3);
```

This code sample creates an array of int. The first parameter int.class given to the Array.newInstance() method tells what type each element in the array should be of. The second parameter states how many elements the array should have space for.

9.3 Accessing Arrays

It is also possible to access the elements of an array using Java Reflection. This is done via the Array.get(...) and Array.set(...) methods. Here is an example:

```
int[] intArray = (int[]) Array.newInstance(int.class, 3);
Array.set(intArray, 0, 123);
Array.set(intArray, 1, 456);
Array.set(intArray, 2, 789);
System.out.println("intArray[0] = " + Array.get(intArray, 0));
System.out.println("intArray[1] = " + Array.get(intArray, 1));
System.out.println("intArray[2] = " + Array.get(intArray, 2));
```

This code sample will print out this:

```
intArray[0] = 123
intArray[1] = 456
intArray[2] = 789
```

9.4 Obtaining the Class Object of an Array

- Using non-reflection code you can do like this:

  Class stringArrayClass = String[].class;

- Doing this using Class.forName() is not quite straightforward. For instance,

  you can access the primitive int array class object like this:

  ```
  Class intArray = Class.forName("[I");
  ```

  The JVM represents an int via the letter I. The [on the left means it is the class of an int array I am interested in. This works for all other primitives too.

  For objects you need to use a slightly different notation:

  ```
  Class stringArrayClass = Class.forName("[Ljava.lang.String;");
  ```

  Notice the [L to the left of the class name, and the ; to the right. This means an array of objects with the given type.

- obtain the class object of primitives

  NOTE: you cannot obtain the class object of primitives using Class.forName(). Both of the examples below result in a ClassNotFoundException:

  Class intClass1 = Class.forName("I");
  Class intClass2 = Class.forName("int");
  I usually do something like this to obtain the class name for primitives as well as objects:

```
public Class getClass(String className){
        if("int" .equals(className)) return int .class;
        if("long".equals(className)) return long.class;
         ...
        return Class.forName(className);
}
```

Once you have obtained the Class object of a type there is a simple way to obtain the Class of an array of that type. The solution, or workaround as you might call it, is to create an empty array of the desired type and obtain the class object from that empty array. It's a bit of a cheat, but it works. Here is how that looks:

```
Class theClass = getClass(theClassName);
Class stringArrayClass = Array.newInstance(theClass, 0).getClass();
```

This presents a single, uniform method to access the array class of arrays of any type. No fiddling with class names etc.

To make sure that the Class object really is an array, you can call the Class.isArray() method to check:

```
Class stringArrayClass = Array.newInstance(String.class, 0).getClass();
System.out.println("is array: " + stringArrayClass.isArray());
```

9.5 Obtaining the Component Type of an Array

Once you have obtained the Class object for an array you can access its component type via the Class.getComponentType() method. The component type is the type of the items in the array. For instance, the component type of an int[] array is the int.class Class object. The component type of a String[] array is the java.lang.String Class object.

Here is an example of accessing the component type array:

```
String[] strings = new String[3];
Class stringArrayClass = strings.getClass();
Class stringArrayComponentType = stringArrayClass.getComponentType();
System.out.println(stringArrayComponentType);
```

This example will print out the text "java.lang.String" which is the component type of the String array.