**Overview of Reflection**

*Java Reflection* is a process of examining or modifying the runtime behavior of a class at runtime.

*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.

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.

This guide will get into Java reflection in depth. It will explain the basics of Java Reflection for classes, methods, fields, constructors, enums, useful Reflection Utility methods and Reflection Utility class.

**Drawbacks of Reflection**

*Reflection* is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The following concerns should be kept in mind when accessing code via reflection.

**1. Performance Overhead**

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

**2. 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.

**3. 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. The reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

* The common uses of reflection for accessing and manipulating classes, fields, methods, and constructors.
* The *java.lang.Class* provides many methods that can be used to get metadata, examine and change the runtime behavior of a class.
* The *java.lang* and *java.lang.reflect* packages provide classes for java reflection.

In this guide, we will learn the reflection for accessing and manipulating classes, fields, methods, and constructors.

**Where it is used**

The Reflection API is mainly used in:

* IDE (Integrated Development Environment) e.g. Eclipse, MyEclipse, NetBeans etc.
* Debugger
* Test Tools etc.

Some of the frameworks that use java reflection are:

* JUnit – uses reflection to parse *@Test* annotation to get the test methods and then invoke it.
* Spring – dependency injection, read more at Spring Dependency Injection
* Tomcat web container to forward the request to correct module by parsing their web.xml files and request URI.
* Eclipse auto-completion of method names
* Struts
* Hibernate

**Java Reflection API Guide**

This guide will get into Java reflection in depth. It will explain the basics of Java Reflection for classes, methods, fields, constructors, enums, useful Reflection Utility methods and Reflection Utility class. This guide covers all the Java reflection concepts with examples.

**Java Reflection for Classes**

These are Java Reflection APIs to get metadata of Class.

* Get Class Object
* Get Super Class
* Get Public Member Classes
* Get Declared Classes
* Get Declaring Class
* Getting Package Name
* Getting Class Modifiers
* Get Type Parameters
* Get Implemented Interfaces
* Get All Public Methods
* Get All Public Constructors
* Get All Public Fields
* Get All Annotations

**Java Reflection for Methods**

The **java.lang.reflect.Method** class provides APIs to access information about a method's modifiers, return type, parameters, annotations, and thrown exceptions. It also be used to invoke methods.

* Obtaining Method Type Information
* Obtaining Names of Method Parameters
* Retrieving and Parsing Method Modifiers
* Get Public Method
* Invoking Public Method
* Get All Public Methods

**Java Reflection for Fields**

Reflection API provides several methods to analyze Class fields and modify their values at runtime, in this guide, we will look into some of the commonly used reflection functions for methods.

* Get All Public Fields
* Get Public Field
* Field Declaring Class
* Get Field Type
* Get/Set Public Field Value
* Get/Set Private Field Value

**Java Reflection for Constructors**

A constructor is used in the creation of an object that is an instance of a class. Typically it performs operations required to initialize the class before methods are invoked or fields are accessed. Constructors are never inherited.

* Finding Constructors
* Retrieving and Parsing Constructor Modifiers
* Creating New Class Instances
* Get Constructor
* Get Constructor Information

**Java Reflection for Arrays**

Arrays have a component type and a length (which is not part of the type). Arrays may be manipulated either in their entirety or component by component. Reflection provides the **java.lang.reflect.Array** class.

* *Identifying Array Types* - describes how to determine if a class member is a field of array type
* *Creating New Arrays* - illustrates how to create new instances of arrays with simple and complex component types
* *Getting and Setting Arrays and Their Components* - shows how to access fields of type array and individually access array elements

**25 Java Reflection Utility Methods**

1. Find Field

2. Set Field

3. Get Field

4. Make the Given Field Accessible

5. Check Field "public static final" constant

6. Get Fields Including Super Classes

7. Find Method

8. Invoke the specified Method

9. Check method is an "equals" method

10. Check method is an "hashCode" method

11. Check method is an "toString" method

12. Make the Given Method Accessible

13. Get All Setter Methods

14. Check if Class is Public

15. Check if Method or Field or Constructor is Public

16. Check If Class is Private

17. Check If Method or Field or Constructor is Private

18. Check if Method or Field or Constructor is not Private

19. Check if the Class is Abstract

20. Check if Method or Field or Constructor Abstract

21. Check if the Class is Static

22. Check if Method or Field or Constructor are static

23. Check if the Class is Inner Class

24. Check if the Object is an Array

25. Make the Given Constructor Accessible

## 1. Overview

In Java, every type is either a reference or a primitive. *Classes*, *enums*, and *arrays* (which all inherit from java.lang.Object), as well as interfaces, are all reference types. Examples of reference types include *java.lang.String*, all of the wrapper classes for primitive types such as *java.lang.Double*, the interface *java.io.Serializable*, and the enum *javax.swing.SortOrder*. There is a fixed set of primitive types: *boolean, byte, short, int, long, char, float, and double*.

*java.lang.Class* is the entry point for all the reflection operations. For every type of object, JVM instantiates an immutable instance of *java.lang.Class* that provides methods to examine the runtime properties of the object and create new objects, invoke its method and get/set object fields.  
  
The source code for this post is available on **GitHub**.  
  
The below class diagram shows a list of Reflection APIs offered by **java.lang.Class**.

|  |
| --- |
| **https://1.bp.blogspot.com/-Y7v0suOMCs4/W17LbMA3aPI/AAAAAAAAC60/1onG10ww8iooCmwAf-P5pUBht_BdsqxywCLcBGAs/s1600/reflection-class.gif** |
| *Java Reflection for Classes* |

## 2. Reflection Class API Examples

* Get Class Object
* Get Super Class
* Get Public Member Classes
* Get Declared Classes
* Get Declaring Class
* Getting Package Name
* Getting Class Modifiers
* Get Type Parameters
* Get Implemented Interfaces
* Get All Public Methods
* Get All Public Constructors
* Get All Public Fields
* Get All Annotations

In this article, we will look into all the above important methods of Class. Let's first create *BaseInterface*, *BaseClass*and *ConcreteClass*then we will apply all the reflection API's on these reference types.

### BaseInterface.java

package com.javaguides.reflection.classes;

public interface BaseInterface {

void method1();

void method2(String str);

void method3(int a);

}

### BaseClass.java

package com.javaguides.reflection.classes;

public class BaseClass {

public void method4(){

System.out.println("method4");

}

public void method5(){

System.out.println("method5");

}

public void method6(){

System.out.println("method6");

}

}

### ConcreteClass.java

package com.javaguides.reflection.classes;

@Deprecated

public class ConcreteClass extends BaseClass implements BaseInterface {

public int id;

private String name;

public ConcreteClass(int id){

this.id = id;

}

public void method1() {

System.out.println("ConcreteClass :: method 1");

}

public void method2(String str) {

System.out.println("ConcreteClass :: method 2");

}

public void method3(int a) {

System.out.println("ConcreteClass :: method 3");

}

// Inner classes

private class InnerPrivateClass {

}

public class InnerPublicClass {

}

protected class InnerProtectedClass {

}

class InnerDefaultClass {

}

// Member enum

public enum ConcreClassMemberEnum {

}

// Member Interface

public interface ConcreteMemberInterface {

}

}

## 2.1 Get Class Object

There are 3 ways to get the instance of Class class. They are as follows:

* *forName()*method of Class class
* *getClass()* method of Object class
* the *.class* syntax

Examples to demonstrate all 3 ways to get the instance of Class class.

Class<BaseClass> baseClass = BaseClass.class;

System.out.println(baseClass.getCanonicalName());

// using forName() method

// Returns the Class object associated with the class or interface with

// the given string name

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

System.out.println(concreteClass.getSimpleName());

// for primitive types

Class<?> booleanClass = boolean.class;

System.out.println(booleanClass.getCanonicalName()); // prints boolean

// wrapper classes

Class<?> cDouble = Double.TYPE;

System.out.println(cDouble.getCanonicalName()); // prints double

// For arrays

Class<?> cDoubleArray = Class.forName("[D");

System.out.println(cDoubleArray.getCanonicalName()); // prints double[]

// Two Dimensional array.

Class<?> twoDStringArray = String[][].class;

System.out.println(twoDStringArray.getCanonicalName()); // prints

//

java.lang.String[][]

// For Collections

Set<String> s = new HashSet<String>();

Class c4 = s.getClass();

System.out.println(c4.getCanonicalName());

// For String

Class c = "foo".getClass();

System.out.println(c.getCanonicalName());

Output:

com.javaguides.reflection.classes.BaseClass

ConcreteClass

boolean

double

double[]

java.lang.String[][]

java.util.HashSet

java.lang.String

## 2.2 Get Super Class

*getSuperclass()* method on a Class object returns the superclass of the class. If this *Class* represents either the *Object* class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the *Class* object representing the *Object* class is returned.

// Get super class BaseClass of ConcreteClass.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

System.out.println(concreteClass.getSuperclass());

// Get super class of Object class

System.out.println(Object.class.getSuperclass()); // prints null

// Get super class of String class

System.out.println(String.class.getSuperclass());

// Get super class of String[]

System.out.println(String[].class.getSuperclass());

// Get Super class of Integer class

System.out.println(Integer.class.getSuperclass());

Output:

class com.javaguides.reflection.classes.BaseClass

null

class java.lang.Object

class java.lang.Object

class java.lang.Number

## 2.3 Get Public Member Classes

*getClasses()* method of a Class representation of object returns an array containing Class objects representing all the public classes, interfaces and enums that are members of the class represented by this Class object. This includes public class and interface members inherited from superclasses and public class and interface members declared by the class. This method returns an array of length 0 if this Class object has no public member classes or interfaces or if this Class object represents a primitive type, an array class, or void.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

Class<?>[] classes = concreteClass.getClasses();

for (Class<?> class1 : classes) {

System.out.println(class1.getCanonicalName());

}

Output:

com.javaguides.reflection.classes.ConcreteClass.ConcreClassMemberEnum

com.javaguides.reflection.classes.ConcreteClass.ConcreteMemberInterface

com.javaguides.reflection.classes.ConcreteClass.InnerPublicClass

## 2.4 Get Declared Classes

*getDeclaredClasses()* method returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object. The returned array doesn’t include classes declared in inherited classes and interfaces.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

Class<?>[] classes = concreteClass.getDeclaredClasses();

for (Class<?> class1 : classes) {

System.out.println(class1.getCanonicalName());

}

Output:

com.javaguides.reflection.classes.ConcreteClass.ConcreClassMemberEnum

com.javaguides.reflection.classes.ConcreteClass.ConcreteMemberInterface

com.javaguides.reflection.classes.ConcreteClass.InnerDefaultClass

com.javaguides.reflection.classes.ConcreteClass.InnerPrivateClass

com.javaguides.reflection.classes.ConcreteClass.InnerProtectedClass

com.javaguides.reflection.classes.ConcreteClass.InnerPublicClass

## 2.6 Get Declaring Class

*getDeclaringClass()* method returns the Class object representing the class in which it was declared.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass$InnerProtectedClass");

System.out.println(concreteClass.getDeclaringClass().getCanonicalName());

// prints com.javaguides.reflection.classes.ConcreteClass

Output:

com.javaguides.reflection.classes.ConcreteClass

## 2.7 Getting Package Name

*getPackage()* method returns the package for this class. The class loader of this class is used to find the package. We can invoke getName() method of Package to get the name of the package.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

System.out.println(concreteClass.getPackage());

Output:

package com.javaguides.reflection.classes

## 2.8 Getting Class Modifiers

*getModifiers()* method returns the int representation of the class modifiers, we can use *java.lang.reflect.Modifier.toString()* method to get it in the string format as used in source code.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

System.out.println(Modifier.toString(concreteClass.getModifiers()));

// prints public

Output:

public

## 2.9 Get Type Parameters

*getTypeParameters()* returns the array of *TypeVariable* if there are any Type parameters associated with the class. The type parameters are returned in the same order as declared.

//Get Type parameters (generics)

TypeVariable<?>[] typeParameters = Class.forName("java.util.HashMap").getTypeParameters();

for(TypeVariable<?> t : typeParameters)

System.out.print(t.getName()+",");

## 2.10 Get Implemented Interfaces

*getGenericInterfaces()* method returns the array of interfaces implemented by the class with generic type information. We can also use *getInterfaces()* to get the class representation of all the implemented interfaces.

Type[] interfaces = Class.forName("java.util.HashMap").getGenericInterfaces();

// prints "[java.util.Map<K, V>, interface java.lang.Cloneable,

// interface java.io.Serializable]"

System.out.println(Arrays.toString(interfaces));

// prints "[interface java.util.Map, interface java.lang.Cloneable,

// interface java.io.Serializable]"

System.out.println(Arrays.toString(Class.forName("java.util.HashMap").getInterfaces()));

Output:

[java.util.Map<K, V>, interface java.lang.Cloneable, interface java.io.Serializable]

[interface java.util.Map, interface java.lang.Cloneable, interface java.io.Serializable]

## 2.11 Get All Public Methods

*getMethods()* method returns the array of public methods of the Class including public methods of it’s superclasses and super interfaces.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

Method[] methods = concreteClass.getMethods();

for (Method method : methods) {

System.out.println(method.getName());

}

Output:

method1

method2

method3

method4

method5

method6

wait

wait

wait

equals

toString

hashCode

getClass

notify

notifyAll

## 2.12 Get All Public Constructors

*getConstructors()* method returns the list of public constructors of the class reference of the object.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

Constructor<?>[] constructors = concreteClass.getConstructors();

for (Constructor constructor : constructors) {

System.out.println(constructor.getName());

}

Output:

com.javaguides.reflection.classes.ConcreteClass

## 2.13 Get All Public Fields

*getFields()* method returns the array of public fields of the class including public fields of its superclasses and super interfaces.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

Field[] fields = concreteClass.getFields();

for (Field field : fields) {

System.out.println(field.getName());

}

Output:

id

## 2.14 Get All Annotations

*getAnnotations()* method returns all the annotations for the element, we can use it with class, fields, and methods also.

Class<?> concreteClass = Class.forName("com.javaguides.reflection.classes.ConcreteClass");

Annotation[] annotations = concreteClass.getAnnotations();

for (Annotation annotation : annotations) {

System.out.println(annotation);

}

Output:

@java.lang.Deprecated()