http://www.java2s.com/Tutorial/Java/0125\_\_Reflection/Catalog0125\_\_Reflection.htm

Using java reflection we can inspect a class, [interface](https://www.journaldev.com/1601/interface-in-java), [enum](https://www.journaldev.com/716/java-enum), get their structure, methods and fields information at runtime even though class is not accessible at compile time. We can also use reflection to instantiate an object, invoke it’s methods, change field values.

**Some of the frameworks that use java reflection are:**

1. JUnit – uses reflection to parse @Test annotation to get the test methods and then invoke it.
2. Spring – [dependency injection](https://www.journaldev.com/2394/java-dependency-injection-design-pattern-example-tutorial)
3. Tomcat web container to forward the request to correct module by parsing their web.xml files and request URI.
4. Eclipse auto completion of method names
5. [Struts](https://www.journaldev.com/dev/struts-2)
6. Hibernate

They all use java reflection because all these frameworks have no knowledge and access of user defined classes, interfaces, their methods etc. For instance, Java Reflection can be used to map properties in JSON files to getter / setter methods in Java objects, like [Jackson, GSON, Boon etc.](http://tutorials.jenkov.com/java-json/index.html) does. Or, Reflection can be used to map the column names of a [JDBC](http://tutorials.jenkov.com/jdbc/index.html) ResultSet to getter / setter methods in a Java object.

We should not use reflection in normal programming where we already have access to the classes and interfaces because of following drawbacks.

* Poor Performance – Since java reflection resolves the types dynamically, it involves processing like scanning the classpath to find the class to load, causing slow performance.
* Security Restrictions – Reflection requires runtime permissions that might not be available for system running under security manager. This can cause you application to fail at runtime because of security manager.
* Security Issues – Using reflection we can access part of code that we are not supposed to access, for example we can access private fields of a class and change it’s value. This can be a serious security threat and cause your application to behave abnormally.
* High Maintenance – Reflection code is hard to understand and debug, also any issues with the code can’t be found at compile time because the classes might not be available, making it less flexible and hard to maintain.

**java.lang.Class** is the entry point for all the reflection operations. For every type of object, [JVM](https://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm) instantiates an [immutable](https://www.journaldev.com/129/how-to-create-immutable-class-in-java) 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.

**Get class object**

We can get Class of an object using three methods – through static variable class, using getClass() method of object and java.lang.Class.forName(String fullyClassifiedClassName). For primitive types and arrays, we can use static variable class. Wrapper classes provide another static variable TYPE to get the class. The Class.forName() method may throw a ClassNotFoundException if the class cannot be found on the classpath at runtime.

**Class object:**

If you want the class name without the pacakge name you can obtain it using the getSimpleName() method, else getName() can be used.

aClass.getPackage();

Class c= aClass.getSuperclass();

Class[] interfaces = aClass.getInterfaces();

Constructor[] constructors = aClass.getConstructors();

Method[] method = aClass.getMethods();

Field[] method = aClass.getFields();

Annotation[] annotations = aClass.getAnnotations();

Only the interfaces specifically declared implemented by a given class is returned. If a superclass of the class implements an interface, but the class doesn't specifically state that it also implements that interface, that interface will not be returned in the array. Even if the class in practice implements that interface, because the superclass does.

**Modifier class**

You can access the modifiers of a class via the Class object. The class modifiers are the keywords "public", "private", "static" etc. You obtain the class modifiers like this:

Class aClass = ... //obtain Class object.

int modifiers = aClass.getModifiers();

The modifiers are packed into an int where each modifier is a flag bit that is either set or cleared. You can check the modifiers using these methods in the class java.lang.reflect.Modifier

Modifier.isAbstract(int modifiers)

Modifier.isFinal(int modifiers)

Modifier.isInterface(int modifiers)

Modifier.isNative(int modifiers)

Modifier.isPrivate(int modifiers)

Modifier.isProtected(int modifiers)

Modifier.isPublic(int modifiers)

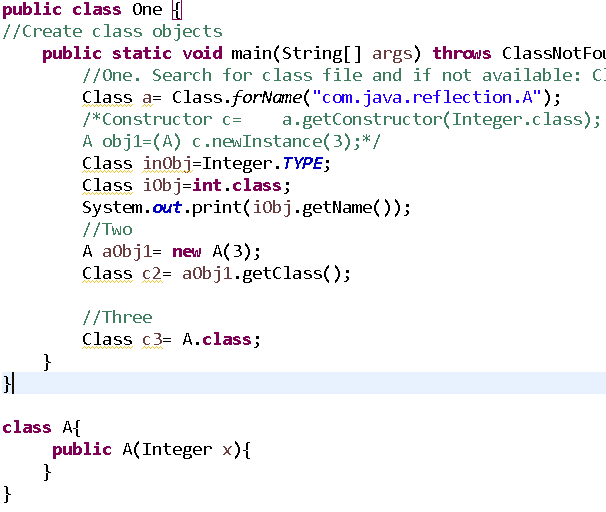
Modifier.isStatic(int modifiers)

Modifier.isStrict(int modifiers)

Modifier.isSynchronized(int modifiers)

Modifier.isTransient(int modifiers)

Modifier.isVolatile(int modifiers)



Q) Get classes method

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.

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.

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.

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.

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

If you know the precise parameter types of the method you want to access, you can do so rather than obtain the array all methods. This example returns the public method named "doSomething", in the given class which takes a String as parameter:

Class aClass = ...//obtain class object

Method method =

aClass.getMethod("doSomething", new Class[]{String.class});

If no method matches the given method name and arguments, in this case String.class, a NoSuchMethodException is thrown.

If the method you are trying to access takes no parameters, pass null as the parameter type array, like this:

Class aClass = ...//obtain class object

Method method =

aClass.getMethod("doSomething", null);

**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. In this example, if doSomething(String.class) is not static, you need 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.

There has been a lot of talk about disabling the ability to access private fields via reflection from Java 9.

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

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 String as 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, a NoSuchMethodException is thrown.

We can read what parameters a given constructor takes like this:

Constructor constructor = ... // obtain constructor - see above

Class[] parameterTypes = constructor.getParameterTypes();

**Create object from constructor:**

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.

**To call private constructor**



**Fields info:**

getFields() method returns the array of public fields of the class including public fields of it’s super classes and super interfaces.

If you know the name of the field you want to access, you can access it like this:

Class aClass = MyObject.class

Field field = aClass.getField("someField");

The example above will return the Field instance corresponding to the field someField as declared in the MyObject below:

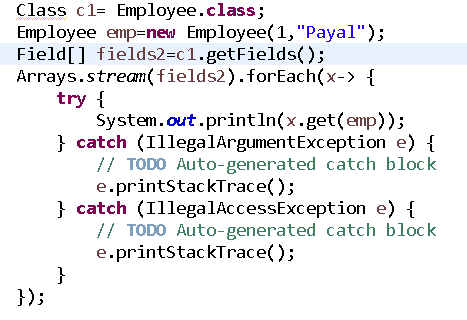
public class MyObject{

public String someField = null;

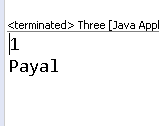
}

If no field exists with the name given as parameter to the getField() method, a NoSuchFieldException is thrown.

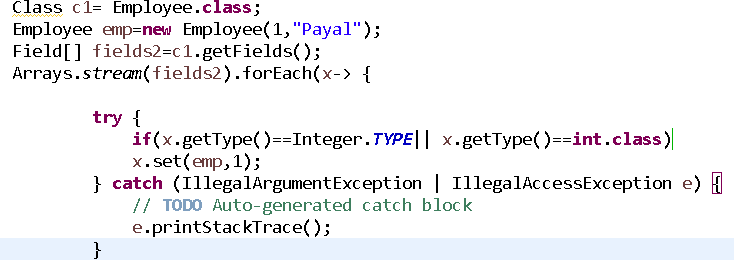
Once you have obtained a Field reference you can get and set its values using the Field.get() and Field.set()methods, only for public fields like this



o/p:

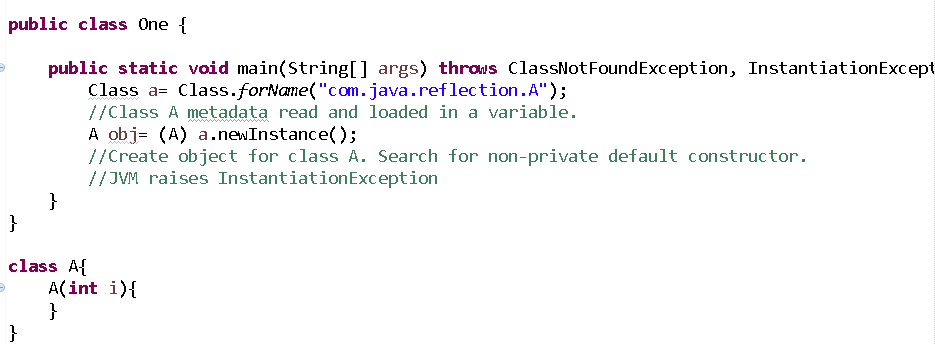


**Set field value:**

****

The objectInstance parameter passed to the get and set method should be an instance of the class that owns the field. It the field is a static field (public static ...) pass null as parameter to the get and set methods, instead of the objectInstance parameter passed

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

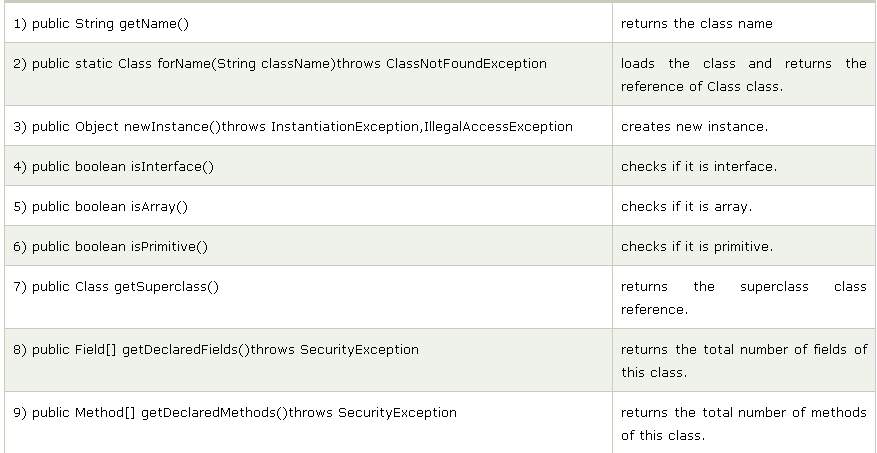


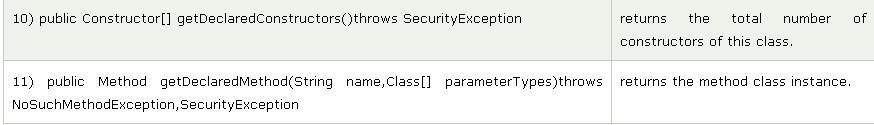
Compiler, JVM, servers, frameworks uses reflection api to read internal details of class.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

Spring framework also uses reflection to create objects of bean classes.

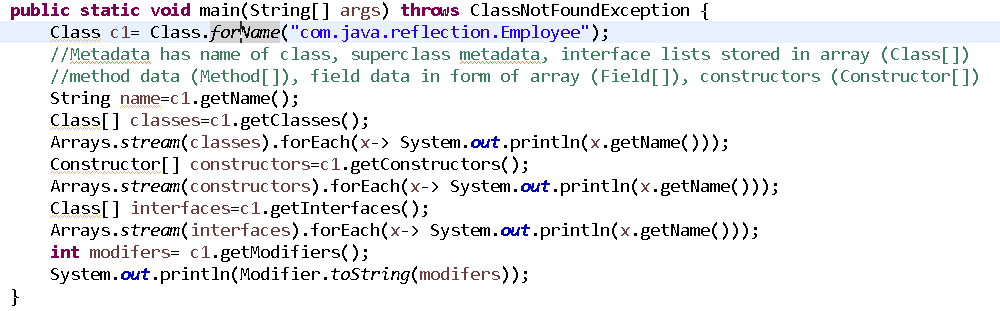


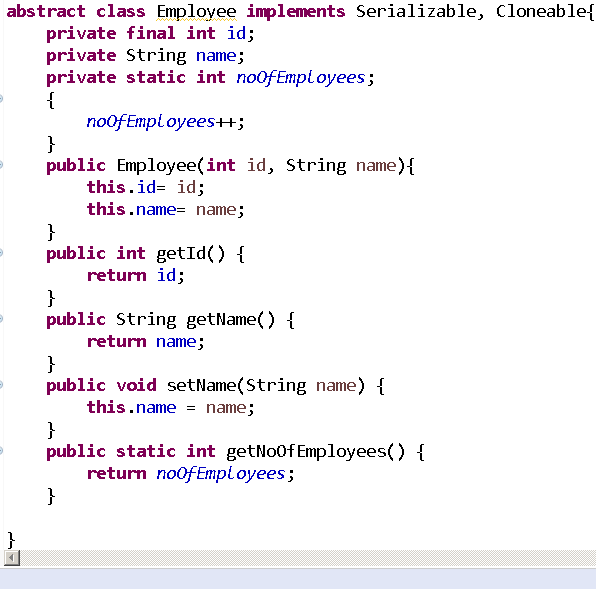


**Various classes:**

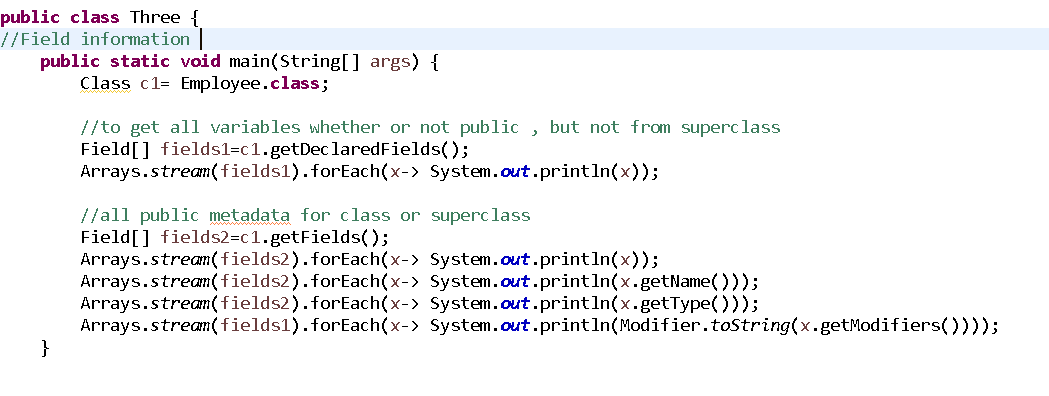
1. Java.lang.Class
2. Java.lang.reflect.Field: to get declarative info about field, type , modifier of variable
3. Java.lang.reflect.Method: metadata of method. Data type, access modifier, name, ..
4. Java.lang.reflect.Constructor
5. Java.lang.reflect.Modifier

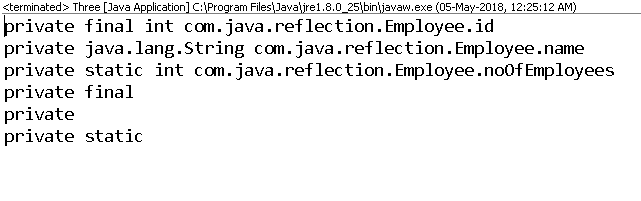
Q) To get metadata of class Employee



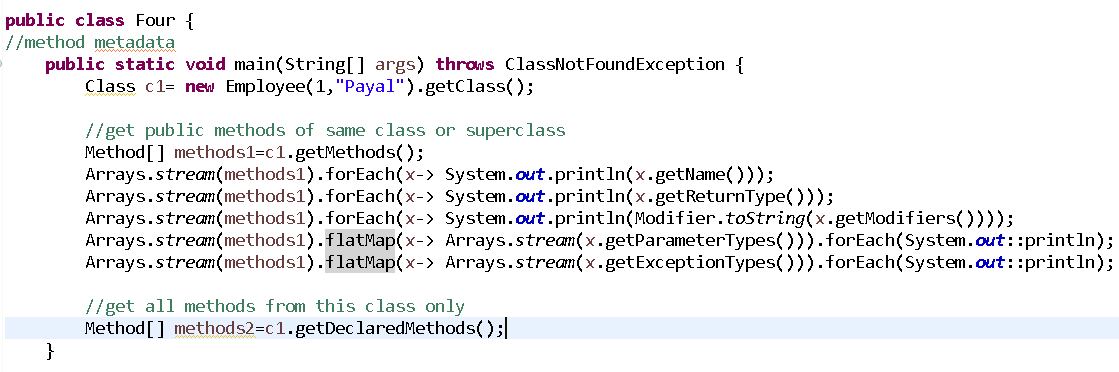


Q) To get metadata of variables:





Q) Methods information of Employee class:



o/p:

getName

getId

setName

getNoOfEmployees

wait

wait

wait

equals

toString

hashCode

getClass

notify

notifyAll

class java.lang.String

int

void

int

void

void

void

boolean

class java.lang.String

int

class java.lang.Class

void

void

public

public

public

public static

public final

public final

public final native

public

public

public native

public final native

public final native

public final native

class java.lang.String

long

int

long

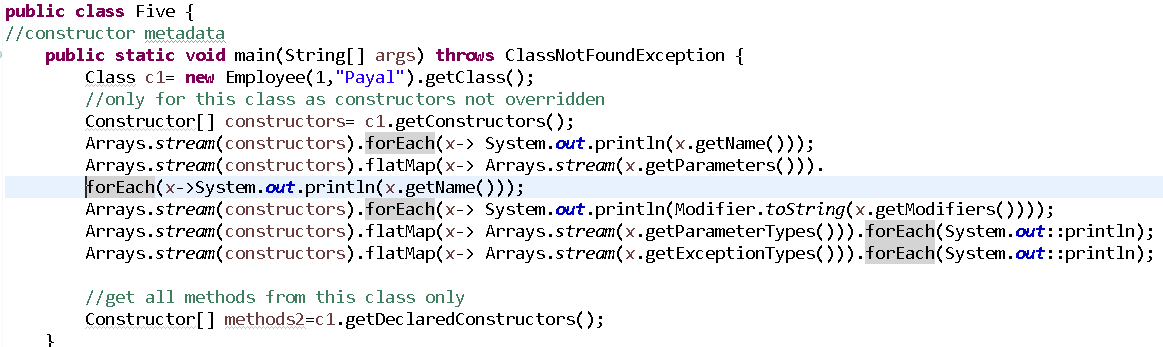
class java.lang.Object

class java.lang.InterruptedException

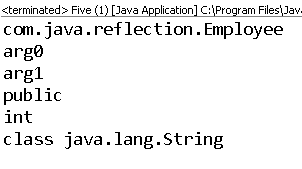
class java.lang.InterruptedException

class java.lang.InterruptedException

Q) Constructor metadata”



o/p:



## Class.forName()

* By default the classes are initialized at the time of loading. It means that static variables in the classes are initialized.
* Also the class is loaded from the current class loader. When you invoke the Class.forName for loading the JDBC driver class, it is loaded to the same class loader from where it is invoked. In short, it is loaded to the caller’s class loader.
* Class.forName is overloaded method. Invoking with single string parameter is equivalent of Class.forName(className, true, currentLoader). Optionally you can pass the second and third parameters to change the behavior.
  + className – Fully qualified name of the class to be loaded
  + initialize – Whether to initialize the class or not. By default the value is “true”
  + classLoader – By default the value is current class loader. Optionally you can change the class loader name.

## ClassLoader.loadClass()

* By default, the classes are not initialized. The classes are loaded and made available in the classpath, the variables are initialized only when it is first time invoked by the caller.
* Another advantage of this class is that you can load the classes to any specific class loader. Which may or may not be the loader that loads that calling code. If picking a specific loader to load the class is important to your design, you should ClassLoader.loadClass().

## Class Loading Example

**TestClass.java**

package javabeat.net.corejava;

public class TestClass {

static {

System.out.println("Static Initializer Called!!");

}

}

**ClassLoadingExample.java**

package javabeat.net.corejava;

public class ClassLoadingExample {

public static void main(String[] args) {

try {

System.out.println("Before Loading the forName");

Class.forName("javabeat.net.corejava.TestClass");

System.out.println("After Loading the forName");

ClassLoader.getSystemClassLoader().loadClass("javabeat.net.corejava.TestClass");

System.out.println("After Loading the loadClass");

}catch (ClassNotFoundException e){

e.printStackTrace();

}

}

}

**Output**

Before Loading the forName

Static Initializer Called!!

After Loading the forName

After Loading the loadClass

## ClassNotFoundException

If the class is not found in the classpath, you would encounter the following exception.

We can see the classes that got loaded using -verbose:class as the VM arguments

