Java Reflection

What is java Reflection?

**Java Reflection** provides ability to inspect and modify the runtime behavior of application. Reflection in Java is one of the advance topic of core java. 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" \o "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 Example :

|  |  |
| --- | --- |
| **This class name is BaseInterface.java**  **package** com.journaldev.reflection.example;  **public** **interface** BaseInterface {  **public** **int** ***interfaceInt*** = 0;  **void** method1();  **int** method2(String str);  } | This is a interface class. It has one variable and two methods. |

|  |  |
| --- | --- |
| **This class name is BaseClass.java**  **package** com.journaldev.reflection.example;  **public** **class** BaseClass {  **public** **int** baseInt;    **private** **static** **void** method3(){  System.***out***.println("Method3");  }    **public** **int** method4(){  System.***out***.println("Method4");  **return** 0;  }    **public** **static** **int** method5(){  System.***out***.println("Method5");  **return** 0;  }    **void** method6(){  System.***out***.println("Method6");  }    // inner public class  **public** **class** BaseClassInnerClass{}    //member public enum  **public** **enum** BaseClassMemberEnum{}  } | This is a java class. |

|  |  |
| --- | --- |
| **This class name is ConcreteClass.java**  **package** com.journaldev.reflection.example;  @Deprecated  **public** **class** ~~ConcreteClass~~ **extends** BaseClass **implements** BaseInterface {  **public** **int** publicInt;  **private** String privateString="private string";  **protected** **boolean** protectedBoolean;  Object defaultObject;    **public** ConcreteClass(**int** i){  **this**.~~publicInt~~=i;  }  @Override  **public** **void** method1() {  System.***out***.println("Method1 impl.");  }  @Override  **public** **int** method2(String str) {  System.***out***.println("Method2 impl.");  **return** 0;  }    @Override  **public** **int** method4(){  System.***out***.println("Method4 overriden.");  **return** 0;  }    **public** **int** method5(**int** i){  System.***out***.println("Method5 overriden.");  **return** 0;  }  // inner classes  **public** **class** ConcreteClassPublicClass{}  **private** **class** ConcreteClassPrivateClass{}  **protected** **class** ConcreteClassProtectedClass{}  **class** ConcreteClassDefaultClass{}    //member enum  **enum** ConcreteClassDefaultEnum{}  **public** **enum** ConcreteClassPublicEnum{}    //member interface  **public** **interface** ConcreteClassPublicInterface{}  } |  |

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.

|  |
| --- |
| **This class name is Test1.java**  **package** com.journaldev.reflection.example;  **public** **class** Test1 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  //get class using reflection  Class<?> concreteClass=~~ConcreteClass~~.**class**;  concreteClass=**new** ~~ConcreteClass~~(5).getClass();  **try** {  // below method is used most of the times in frameworks like JUnit  //Spring dependency injection, Tomcat web container  //Eclipse auto completion of method names, hibernate, Struts2 etc.  //because ConcreteClass is not available at compile time  concreteClass=Class.*forName*("com.journaldev.reflection.example.ConcreteClass")  }  **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  System.***out***.println(concreteClass.getCanonicalName());    // for primitives types, wrapper classes and arrays  Class<?> booleanClass=**boolean**.**class**;  System.***out***.println(booleanClass.getCanonicalName());    Class<?> cDouble=Double.***TYPE***;  System.***out***.println(cDouble.getCanonicalName());    Class<?> cDoubleArray = Class.*forName*("[D");  System.***out***.println(cDoubleArray.getCanonicalName());    Class<?> twoStringArray=String[][].**class**;  System.***out***.println(twoStringArray.getCanonicalName());  }  } |
| **Output:**  com.journaldev.reflection.example.ConcreteClass  boolean  double  double[]  java.lang.String[][] |

|  |  |
| --- | --- |
| **This class name is Test2.java**  **package** com.journaldev.reflection.example;  **public** **class** Test2 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  Class<?> superClass = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getSuperclass();  System.***out***.println(superClass); // prints "class com.journaldev.reflection.BaseClass"  System.***out***.println(Object.**class**.getSuperclass()); // prints "null"  System.***out***.println(String[][].**class**.getSuperclass());// prints "class java.lang.Object"  }  } | |
| **Output:**  class com.journaldev.reflection.example.BaseClass  null  class java.lang.Object | **getSuperclass()** method on a Class object returns the super class 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. |

|  |
| --- |
| **This class name is Test3.java**  **package** com.journaldev.reflection.example;  **import** java.util.\*;  **public** **class** Test3 {  **public** **static** **void** main(String[] args) {    Class<?> concreteClass=~~ConcreteClass~~.**class**;  Class<?>[] classes = concreteClass.getClasses();  //[class com.journaldev.reflection.ConcreteClass$ConcreteClassPublicClass,  //class com.journaldev.reflection.ConcreteClass$ConcreteClassPublicEnum,  //interface com.journaldev.reflection.ConcreteClass$ConcreteClassPublicInterface,  //class com.journaldev.reflection.BaseClass$BaseClassInnerClass,  //class com.journaldev.reflection.BaseClass$BaseClassMemberEnum]  **for** (Class<?> class1 : classes) {  System.***out***.println(class1);  }  System.***out***.println("\n------------------------------------------------------");  System.***out***.println(Arrays.*toString*(classes));  }  } |
| **Output:**  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicClass  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicEnum  interface com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicInterface  class com.journaldev.reflection.example.BaseClass$BaseClassInnerClass  class com.journaldev.reflection.example.BaseClass$BaseClassMemberEnum  -------------------------------------------------------------  [class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicClass, class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicEnum, interface com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicInterface, class com.journaldev.reflection.example.BaseClass$BaseClassInnerClass, class com.journaldev.reflection.example.BaseClass$BaseClassMemberEnum] | |
| **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. | |

|  |
| --- |
| **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. |
| **This class name is Test4.java**  **package** com.journaldev.reflection.example;  **import** java.util.\*;  **public** **class** Test4 {  **public** **static** **void** main(String[] args) **throws** SecurityException, ClassNotFoundException {  // getting all of the classes, interfaces, and enums that are explicitly  // declared in ConcreteClass  Class<?>[] explicitClasses = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getDeclaredClasses();  **for** (Class<?> class1 : explicitClasses) {  System.***out***.println(class1);  }  }  } |
| **Output:**  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassDefaultClass  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassDefaultEnum  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPrivateClass  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassProtectedClass  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicClass  class com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicEnum  interface com.journaldev.reflection.example.ConcreteClass$ConcreteClassPublicInterface |

|  |
| --- |
| **Get Declaring Class**  getDeclaringClass() method returns the Class object representing the class in which it was declared. |
| **This class name is Test5.java**  **package** com.journaldev.reflection.example;  **public** **class** Test5 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  Class<?> innerClass = Class.*forName*("com.journaldev.reflection.example.ConcreteClass$ConcreteClassDefaultClass");  //prints com.journaldev.reflection.ConcreteClass  System.***out***.println(innerClass.getDeclaringClass().getCanonicalName());  System.***out***.println(innerClass.getEnclosingClass().getCanonicalName());  }  } |
| **Output:**  com.journaldev.reflection.example.ConcreteClass  com.journaldev.reflection.example.ConcreteClass |

|  |
| --- |
| **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. |
| **This class name is Test6.java**  **package** com.journaldev.reflection.example;  **public** **class** Test6 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  System.***out***.println(Class.*forName*("com.journaldev.reflection.example.BaseInterface").getPackage().getName());  }  } |
| **Output:**  com.journaldev.reflection.example |

|  |
| --- |
| **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. |
| **This class name is Test7.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Modifier;  **public** **class** Test7 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  Class<?> concreteClass=~~ConcreteClass~~.**class**;  System.***out***.println(Modifier.*toString*(concreteClass.getModifiers())); // prints "public"  //prints "public abstract interface"  System.***out***.println(Modifier.*toString*(Class.*forName*("com.journaldev.reflection.example.BaseInterface").getModifiers()));  }  } |
| **Output:**  public  public abstract interface |

|  |
| --- |
| **Get Type Parameters**  getTypeParameters() returns the array of Type Variable if there are any Type parameters associated with the class. The type parameters are returned in the same order as declared. |
| **This class name is Test8.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.TypeVariable;  **public** **class** Test8 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  //Get Type parameters (generics)  TypeVariable<?>[] typeParameters = Class.*forName*("java.util.HashMap").getTypeParameters();  **for**(TypeVariable<?> t : typeParameters)  System.***out***.print(t.getName()+",");  }  } |
| **Output:**  K,V, |

|  |
| --- |
| **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. |
| **This class name is Test9.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Type;  **import** java.lang.reflect.Array;  **import** java.util.Arrays;  **public** **class** Test9 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  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] |

|  |
| --- |
| **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. |
| **This class name is Test10.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Method;  **import** java.util.\*;  **public** **class** Test10 {  **public** **static** **void** main(String[] args) **throws** SecurityException, ClassNotFoundException {  Method[] publicMethods = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getMethods();  //prints public methods of ConcreteClass, BaseClass, Object  **for** (Method method : publicMethods) {  System.***out***.println(method);  }  }  } |
| **Output:**  public void com.journaldev.reflection.example.ConcreteClass.method1()  public int com.journaldev.reflection.example.ConcreteClass.method4()  public int com.journaldev.reflection.example.ConcreteClass.method5(int)  public int com.journaldev.reflection.example.ConcreteClass.method2(java.lang.String)  public static int com.journaldev.reflection.example.BaseClass.method5()  public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException  public final void java.lang.Object.wait() throws java.lang.InterruptedException  public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException  public boolean java.lang.Object.equals(java.lang.Object)  public java.lang.String java.lang.Object.toString()  public native int java.lang.Object.hashCode()  public final native java.lang.Class java.lang.Object.getClass()  public final native void java.lang.Object.notify()  public final native void java.lang.Object.notifyAll() |

|  |
| --- |
| **Get All Public Constructor**  getConstructors() method returns the list of public constructors of the class reference of object. |
| **This class name is Test11.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Constructor;  **import** java.util.\*;  **public** **class** Test11 {  **public** **static** **void** main(String[] args) **throws** SecurityException, ClassNotFoundException {  //Get All public constructors  Constructor<?>[] publicConstructors = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getConstructors();  //prints public constructors of ConcreteClass  **for** (Constructor<?> constructor : publicConstructors) {  System.***out***.println(constructor);  }  System.***out***.println(Arrays.*toString*(publicConstructors));  }  } |
| **Output:**  public com.journaldev.reflection.example.ConcreteClass(int)  [public com.journaldev.reflection.example.ConcreteClass(int)] |

|  |
| --- |
| **Get All Public Fields**  getFields() method returns the array of public fields of the class including public fields of it’s super classes and super interfaces. |
| **This class name is Test12.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Field;  **import** java.util.\*;  **public** **class** Test12 {  **public** **static** **void** main(String[] args) **throws** SecurityException, ClassNotFoundException {  // Get All public fields  Field[] publicFields = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getFields();  // prints public fields of ConcreteClass, it's superclass and super interfaces  **for** (Field field : publicFields) {  System.***out***.println(field);  }  System.***out***.println(Arrays.*toString*(publicFields));  }  } |
| **Output:**  public int com.journaldev.reflection.example.ConcreteClass.publicInt  public static final int com.journaldev.reflection.example.BaseInterface.interfaceInt  public int com.journaldev.reflection.example.BaseClass.baseInt |

|  |
| --- |
| **Get All Annotations**  getAnnotations() method returns all the annotations for the element, we can use it with class, fields and methods also. Note that only annotations available with reflection are with retention policy of RUNTIME |
| **This class name is Test13.java**  **package** com.journaldev.reflection.example;  **import** java.util.\*;  **public** **class** Test13 {  **public** **static** **void** main(String[] args) **throws** SecurityException, ClassNotFoundException {  java.lang.annotation.Annotation[] annotations = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getAnnotations();  //prints [@java.lang.Deprecated()]  System.***out***.println(Arrays.*toString*(annotations));  }  } |
| **Output:**  [@java.lang.Deprecated(forRemoval=false, since="")] |

|  |
| --- |
| **Java Reflection For Fields**  Reflection API provides several methods to analyze Class fields and modify their values at runtime, in this section we will look into some of the commonly used reflection functions for methods.  **Java public Fields**  In last section, we saw how to get the list of all the public fields of a class. Reflection API also provides method to get specific public field of a class through getField() method. This method look for the field in the specified class reference and then in the super interfaces and then in the super classes. |
| **This class name is Test14.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Field;  **public** **class** Test14 {  **public** **static** **void** main(String[] args) **throws** Exception {  Field field =Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getField("interfaceInt");  System.***out***.println(field);  }  } |
| **Output:**  public static final int com.journaldev.reflection.example.BaseInterface.interfaceInt |

|  |
| --- |
| **Field Declaring Class**  We can use getDeclaringClass() of field object to get the class declaring the field. |
| **This class name is Test15.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Field;  **public** **class** Test15 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  **try** {  Field field=Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getField("interfaceInt");  Class<?> fieldClass = field.getDeclaringClass();  System.***out***.println(fieldClass.getCanonicalName());  // prints com.journaldev.reflection.BaseInterface  }  **catch** (NoSuchFieldException | SecurityException e) {  e.printStackTrace();  }  }  } |
| **Output:**  com.journaldev.reflection.example.BaseInterface |

|  |
| --- |
| **Get Field Type**  getType() method returns the Class object for the declared field type, if field is primitive type, it returns the wrapper class object. |
| **This class name is Test16.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Field;  **public** **class** Test16 {  **public** **static** **void** main(String[] args) **throws** NoSuchFieldException, SecurityException,ClassNotFoundException {  Field field = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getField("publicInt");  Class<?> fieldType = field.getType();  System.***out***.println(fieldType.getCanonicalName()); // prints int  }  } |
| **Output:**  int |

|  |
| --- |
| **Get/Set Public Field Value**  We can get and set the value of a field in an Object using reflection. |
| **This class name is Test17.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Field;  **public** **class** Test17 {  **public** **static** **void** main(String[] args) **throws** NoSuchFieldException, SecurityException, ClassNotFoundException, IllegalArgumentException, IllegalAccessException {  Field field Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getField("publicInt");  ~~ConcreteClass~~ obj = **new** ~~ConcreteClass~~(5);  System.***out***.println(field.get(obj)); // prints 5  field.setInt(obj, 10); // setting field value to 10 in object  System.***out***.println(field.get(obj)); // prints 10  }  } |
| get() method return Object, so if field is primitive type, it returns the corresponding [**Wrapper Class**](https://www.journaldev.com/1002/wrapper-class-in-java). If the field is static, we can pass Object as null in get() method.  There are several set\*() methods to set Object to the field or set different types of primitive types to the field. We can get the type of field and then invoke correct function to set the field value correctly. If the field is final, the set() methods throw java.lang.IllegalAccessException. |
| **Output:**  5  10 |

|  |
| --- |
| **Get/Set Private Field Value**  We know that private fields and methods can’t be accessible outside of the class but using reflection we can get/set the private field value by turning off the java access check for field modifiers |
| **This class name is Test18.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Field;  **public** **class** Test18 {  **public** **static** **void** main(String[] args) **throws** Exception{  Field privateField = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getDeclaredField("privateString");  //turning off access check with below method call  privateField.setAccessible(**true**);  ~~ConcreteClass~~ objTest = **new** ~~ConcreteClass~~(1);  System.***out***.println(privateField.get(objTest)); // prints "private string"  privateField.set(objTest, "private string updated");  System.***out***.println(privateField.get(objTest)); // prints "private string updated"  }  } |
| **Output:**  private string  private string updated |

|  |
| --- |
| **Java Reflection For Methods**  Using reflection we can get information about a method and we can invoke it also. In this section, we will learn different ways to get a method, invoke a method and accessing private methods. |
| **Get Public Method**  We can use getMethod() to get a public method of class, we need to pass the method name and parameter types of the method. If the method is not found in the class, reflection API looks for the method in superclass.  In below example, I am getting put() method of HashMap using reflection. The example also shows how to get the parameter types, method modifiers and return type of a method. |
| **This class name is Test19.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Method;  **import** java.util.\*;  **import** java.lang.reflect.Modifier;  **import** java.lang.Object;  **public** **class** Test19 {  **public** **static** **void** main(String[] args) **throws** Exception {  Method method = Class.*forName*("java.util.HashMap").getMethod("put", Object.**class**, Object.**class**);  //get method parameter types, prints "[class java.lang.Object, class java.lang.Object]"  System.***out***.println(Arrays.*toString*(method.getParameterTypes()));  //get method return type, return "class java.lang.Object", class reference for void  System.***out***.println(method.getReturnType());  //get method modifiers  System.***out***.println(Modifier.*toString*(method.getModifiers())); // prints "public"  }  } |
| **Output:**  [class java.lang.Object, class java.lang.Object]  class java.lang.Object  public |

|  |
| --- |
| **Invoking Public Method**  We can use invoke() method of Method object to invoke a method, in below example code I am invoking put method on HashMap using reflection. |
| **This class name is Test20.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Method;  **import** java.util.\*;  **public** **class** Test20 {  **public** **static** **void** main(String[] args) **throws** Exception {  Method method = Class.*forName*("java.util.HashMap").getMethod("put", Object.**class**, Object.**class**);  Map<String, String> hm = **new** HashMap<>();  method.invoke(hm, "key", "value");  System.***out***.println(hm); // prints {key=value}  }  } |
| If the method is static, we can pass NULL as object argument. |
| **Output:**  {key=value} |

|  |
| --- |
| **Invoking Private Method**  We can use getDeclaredMethod() to get the private method and then turn off the access check to invoke it, below example shows how we can invoke method3() of BaseClass that is static and have no parameters. |
| **This class name is Test21.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Method;  **public** **class** Test21 {  **public** **static** **void** main(String[] args) **throws** Exception {  // invoking private method  Method method = Class.*forName*("com.journaldev.reflection.example.BaseClass").getDeclaredMethod  ("method3", **null**);  method.setAccessible(**true**);  method.invoke(**null**, **null**); // prints "Method3"  }  } |
| **Output:**  Method3 |

|  |
| --- |
| **Java Reflection For Constructors**  Reflection API provides methods to get the constructors of a class to analyze and we can create new instances of class by invoking the constructor. We have already learned how to get all the public constructors. |
|  |
| **Get Public Constructor**  We can use getConstructor() method on the class representation of object to get specific public constructor. Below example shows how to get the constructor of ConcreteClass defined above and the no-argument constructor of HashMap. It also shows how to get the array of parameter types for the constructor. |
| **This class name is Test22.java**  **package** com.journaldev.reflection.example;  **import** java.lang.reflect.Constructor;  **import** java.util.\*;  **public** **class** Test22 {  **public** **static** **void** main(String[] args) **throws** Exception {  Constructor<?> constructor = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").getConstructor(**int**.**class**);  //getting constructor parameters  System.***out***.println(Arrays.*toString*(constructor.getParameterTypes())); // prints "[int]"  Constructor<?> hashMapConstructor = Class.*forName*("java.util.HashMap").getConstructor(**null**);  System.***out***.println(Arrays.*toString*(hashMapConstructor.getParameterTypes())); // prints "[]"  }  } |
| **Output:**  [int]  [] |

|  |
| --- |
|  |
|  |
| **Instantiate Object Using Constructor**  We can use newInstance() method on the constructor object to instantiate a new instance of the class. Since we use reflection when we don’t have the classes information at compile time, we can assign it to Object and then further use reflection to access it’s fields and invoke it’s methods. |
| **This class name is Test23.java**  **package** com.journaldev.reflection.example;  **import** java.util.\*;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Method;  **public** **class** Test23 {  **public** **static** **void** main(String[] args) **throws** Exception {  Constructor<?> constructor = Class.*forName*("com.journaldev.reflection.example.ConcreteClass").  getConstructor(**int**.**class**);  //getting constructor parameters  System.***out***.println(Arrays.*toString*(constructor.getParameterTypes())); // prints "[int]"  Object myObj = constructor.newInstance(10);  Method myObjMethod = myObj.getClass().getMethod("method1", **null**);  myObjMethod.invoke(myObj, **null**); // prints "Method1 impl."  Constructor<?> hashMapConstructor = Class.*forName*("java.util.HashMap").getConstructor(**null**);  System.***out***.println(Arrays.*toString*(hashMapConstructor.getParameterTypes())); // prints "[]"  HashMap<String, String> myMap = (HashMap<String, String>) hashMapConstructor.newInstance(**null**);  }  } |
| **Output:**  [int]  Method1 impl.  [] |
|  |