JAVA

Abstract Classes – Reflection – Interfaces – Clone – Lambdas-JDK 5 to JDK 16 Features

Shoiab

Abstract Class

1. Overview

There are many cases when implementing a contract where we want to postpone some parts of the implementation to be completed later. We can easily accomplish this in Java through abstract classes.

In this tutorial, we'll learn the basics of abstract classes in Java, and in what cases they can be helpful.

2. Key Concepts for Abstract Classes

Before diving into when to use an abstract class, **let's look at their most relevant characteristics**:

- **We define an abstract class with the *abstract* modifier preceding the *class* keyword**
- An abstract class can be subclassed, but it can't be instantiated
- If a class defines one or more *abstract* methods, then the class itself must be declared *abstract*
- **An abstract class can declare both abstract and concrete methods**
- A subclass derived from an abstract class must either implement all the base class's abstract methods or be abstract itself

To better understand these concepts, we'll create a simple example.

Let's have our base abstract class define the abstract API of a board game:

3. When to Use Abstract Classes

Now, **let's analyze a few typical scenarios where we should prefer abstract classes over interfaces** and concrete classes:

- We want to encapsulate some common functionality in one place (code reuse) that multiple, related subclasses will share
- We need to partially define an API that our subclasses can easily extend and refine
- The subclasses need to inherit one or more common methods or fields with protected access modifiers

Let's keep in mind that all these scenarios are good examples of full, inheritance-based adherence to the **Open Close Principle**

Moreover, since the use of abstract classes implicitly deals with base types and subtypes, we're also taking advantage of

Note that code reuse is a very compelling reason to use abstract classes, as long as the "is-a" relationship within the class hierarchy is preserved.

A quick Example of using Abstract classes w.r.to Decorator Pattern or Composition and Part-Whole Hierarchy.

```
package hit.day16;
/*
 * What is the Need for Hierarchial Classification
 * 1. Part Whole hierarchy
 * 2. Composition
 * 3. Polymorphic query
 * 4. Object Reusability
 * 5. Removal of If-Else-If
 * 6. Reusability of code
 * The above reasons are the reasons for inheritance.
 *
 * All Parent classes have to abstract
 * - you cannot create a object of parent class because they
are common nouns
 * So in order to ensure that an object should not be
created...we make the class abstract
 */
abstract class Item{
    public abstract int cost();
}
abstract class Cream extends Item{

}
abstract class Ingredients extends Item{

}
```

```
class Vanila extends Cream{//static binding - what cannot be
changed at runtime
	Item item;
	public Vanila() {
		// TODO Auto-generated constructor stub
	}
	public Vanila(Item item) {// dynamic binding - at
constructor level
		this.item=item;
	}
	@Override
	public int cost() {
		if(item==null) {
			return 10;
		}else {
			return 10+item.cost();
		}

	}
}
class ButterScotch extends Cream{
	Item item;
	public ButterScotch() {
		// TODO Auto-generated constructor stub
	}
	public ButterScotch(Item item) {
		this.item=item;
	}
	@Override
	public int cost() {
		// TODO Auto-generated method stub
		if(item==null) {
			return 15;
		}
		else {
			return 15+item.cost();
		}
```

```
	}
}
class Nuts extends Ingredients{
	Item item;
	public Nuts() {

	}
	public Nuts(Item item) {
		this.item=item;
	}
	@Override
	public int cost() {
		// TODO Auto-generated method stub
		if(item==null) {
			return 5;
		}else {
			return 5 +item.cost();
		}
	}
}
class DryFruit extends Ingredients{
	Item item;
	public DryFruit() {
		// TODO Auto-generated constructor stub
	}
	public DryFruit(Item item) {
		// TODO Auto-generated constructor stub
		this.item=item;
	}

	@Override
	public int cost() {
		// TODO Auto-generated method stub
		if(item==null) {
			return 10;
		}
		else {
```

```
                return 10+item.cost();
            }
        }
}

public class AbstractDemo {
     public static void main(String[] args) {
          Item iceCream=new Vanila(new Nuts(new
ButterScotch(new DryFruit(new Vanila()))));
          System.out.println("One Vanila cup..+ Nuts. +
Butter Scotch:"+iceCream.cost());


     }
}
```

Interfaces

A *Java interface* is a bit like a Spcial Class except a Java interface can only contain method signatures and fields. A Java interface is not intended to contain implementations of the methods, only the signature (name, parameters and exceptions) of the method. However, it is possible to provide default implememntations of a method in a Java interface, to make the implementation of the interface easier for classes implementing the interface.

You can use interfaces in Java as a way to achieve polymorphism. I will get back to polymorphism later in this text.

Java Interface Example

Here is a simple Java interface example:

```
public interface MyInterface {

    public String hello = "Hello";

    public void sayHello();
}
```

As you can see, an interface is declared using the Java `interface` keyword. Just like with

classes, a Java interface can be declared `public` or package scope (no access modifier).

The interface example above contains one variable and one method. The variable can be accessed directly from the interface, like this:

```
System.out.println(MyInterface.hello);
```

As you can see, accessing a variable from an interface is very similar to accessing a static variable in a class.

The method, however, needs to be implemented by some class before you can access it. The next section will explain how that is done.

Implementing an Interface

Before you can really use an interface, you must implement that interface in some Java class. Here is a class that implements the `MyInterface` interface shown above:

```
public class MyInterfaceImpl implements MyInterface {

    public void sayHello() {
        System.out.println(MyInterface.hello);
    }
}
```

Notice the `implements MyInterface` part of the above class declaration. This signals to the Java compiler that the `MyInterfaceImpl` class implements the `MyInterface` interface.

A class that implements an interface must implement all the methods declared in the interface. The methods must have the exact same signature (name + parameters) as declared in the interface. The class does not need to implement (declare) the variables of an interface. Only the methods.

Interface Instances

Once a Java class implements an Java interface you can use an instance of that class as an instance of that interface. Here is an example:

```
MyInterface myInterface = new MyInterfaceImpl();

myInterface.sayHello();
```

Notice how the variable is declared to be of the interface type `MyInterface` while the object created is of type `MyInterfaceImpl`. Java allows this because the class `MyInterfaceImpl` implements the `MyInterface` interface. You can then reference instances of the

`MyInterfaceImpl` class as instances of the `MyInterface` interface.

You cannot create instances of a Java interface by itself. You must always create an instance of some class that implements the interface, and reference that instance as an instance of the interface.

Implementing Multiple Interfaces

A Java class can implement multiple Java interfaces. In that case the class must implement all the methods declared in all the interfaces implemented. Here is an example:

```
public class MyInterfaceImpl
    implements MyInterface, MyOtherInterface {

    public void sayHello() {
        System.out.println("Hello");
    }

    public void sayGoodbye() {
        System.out.println("Goodbye");
    }
}
```

This class implements two interfaces called `MyInterface` and `MyOtherInterface`. You list the names of the interfaces to implement after the `implements` keyword, separated by a comma.

If the interfaces are not located in the same packages as the implementing class, you will also need to import the interfaces. Java interfaces are imported using the `import` instruction just like Java classes. For instance:

```
import com.jenkov.package1.MyInterface;
import com.jenkov.package2.MyOtherInterface;

public class MyInterfaceImpl implements MyInterface, MyOtherInterface {
    ...
}
```

Here are the two Java interfaces implemented by the class above:

```
public interface MyInterface {

    public void sayHello();
}
public interface MyOtherInterface {

    public void sayGoodbye();
}
```

As you can see, each interface contains one method. These methods are implemented by the class `MyInterfaceImpl`.

Overlapping Method Signatures

If a Java class implements multiple Java interfaces, there is a risk that some of these interfaces may contain methods with the same signature (name + parameters). Since a Java class can only implement at method with a given signature once, this could potentially lead to some problems.

The Java specification does not give any solution to this problem. It is up to you to decide what to do in that situation.

Interface Constants

A Java interface can contain constants. In some cases it can make sense to define constants in an interface. Especially if those constants are to be used by the classes implementing the interface, e.g. in calculations, or as parameters to some of the methods in the interface. However, my advice to you is to avoid placing variables in Java interfaces if you can.

All variables in an interface are implicitly `public`, `static` and `final`, even if you leave out these keywords in the variable declaration.

Here is an example of a Java interface with two constants defined:

```
public interface MyInterface {

    int FALSE = 0;
    int TRUE  = 1;
}
```

Interface Methods

A Java interface can contain one or more method declarations. As mentioned earlier, the interface cannot specify any implementation for these methods. It is up to the classes implementing the interface to specify an implementation.

All methods in an interface are public, even if you leave out the `public` keyword in the method declaration.

Default Methods

Java 8 has added the concept of *interface default methods* to Java interfaces. An interface default

method can contain a default implementation of that method. Classes that implement the interface but which contain no implementation for the default interface will then automatically get the default method implementation.

You mark a method in an interface as a default method using the `default` keyword. Here is an example of adding a default method to the `ResourceLoader` interface:

```
public interface ResourceLoader {

    Resource load(String resourcePath);

    default Resource load(Path resourcePath) {
        // provide default implementation to load
        // resource from a Path and return the content
        // in a Resource object.
    }

}
```

This example adds the default method `load(Path)`. The example leaves out the actual implementation (inside the method body) because this is not really interesting. What matters is how you declare the interface default method.

A class can override the implementation of a default method simply by implementing that method explicitly, as is done normally when implementing a Java interface. Any implementation in a class takes precedence over interface default method implementations.

Interface Static Methods

A Java interface can have static methods. Static methods in a Java interface must have implementation. Here is an example of a static method in a Java interface:

```
public interface MyInterface {

    public static void print(String text){
        System.out.print(text);
    }
}
```

Calling a static method in an interface looks and works just like calling a static method in a class. Here is an example of calling the static `print()` method from the above `MyInterface` interface:

```
MyInterface.print("Hello static method!");
```

Static methods in interfaces can be useful when you have some utility methods you would like to make available, which fit naturally into an interface related to the same responsibility. For

instance, a `Vehicle` interface could have a `printVehicle(Vehicle v)` static method.

Interfaces and Inheritance

It is possible for a Java interface to inherit from another Java interface, just like classes can inherit from other classes. You specify inheritance using the `extends` keyword. Here is a simple interface inheritance example:

```
public interface MySuperInterface {

    public void saiHello();

}
public interface MySubInterface extends MySuperInterface {

    public void sayGoodbye();
}
```

The interface `MySubInterface` extends the interface `MySuperInterface`. That means, that the `MySubInterface` inherits all field and methods from `MySuperInterface`. That then means, that if a class implements `MySubInterface`, that class has to implement all methods defined in both `MySubInterface` and `MySuperInterface`.

It is possible to define methods in a subinterface with the same signature (name + parameters) as methods defined in a superinterface, should you find that desirable in your design, somehow.

Unlike classes, interfaces can actually inherit from multiple superinterfaces. You specify that by listing the names of all interfaces to inherit from, separated by comma. A class implementing an interface which inherits from multiple interfaces must implement all methods from the interface and its superinterfaces.

Here is an example of a Java interface that inherits from multiple interfaces:

```
public interface MySubInterface extends
    SuperInterface1, SuperInterface2 {

    public void sayItAll();
}
```

As when implementing multiple interfaces, there are no rules for how you handle the situation when multiple superinterfaces have methods with the same signature (name + parameters).

Inheritance and Default Methods

Interface default methods add a bit complexity to the rules of interface inheritance. While it is normally possible for a class to implement multiple interfaces even if the interfaces contain

methods with the same signature, this is not possible if one or more of these methods are default methods. In other words, if two interfaces contain the same method signature (name + parameters) and one of the interfaces declare this method as a default method, a class cannot automatically implement both interfaces.

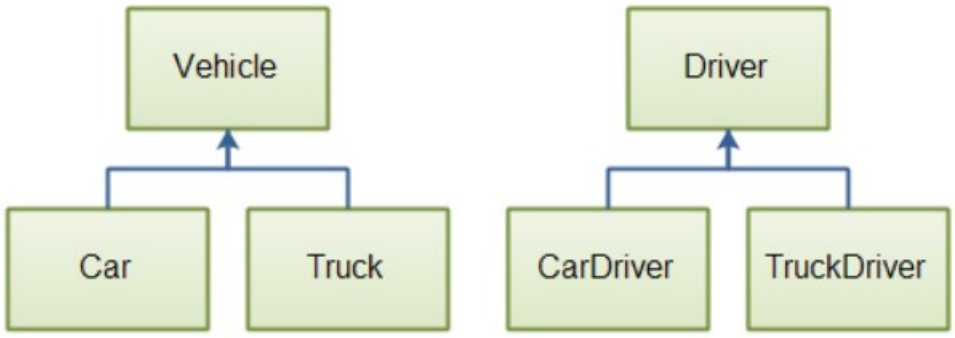
The situation is the same if an interface extends (inherits from) multiple interfaces, and one or more of these interfaces contain methods with the same signature, and one of the superinterfaces declare the overlapping method as a default method.

In both of the above situations the Java compiler requires that the class implementing the interface(s) explicitly implements the method which causes the problem. That way there is no doubt about which implementation the class will have. The implementation in the class takes precedence over any default implementations.

Interfaces and Polymorphism

Java interfaces are a way to achieve polymorphism. Polymorphism is a concept that takes some practice and thought to master. Basically, polymorphism means that an instance of an class (an object) can be used as if it were of different types. Here, a type means either a class or an interface.

Look at this simple class diagram:



Two parallel class hierarchies used in the same application.

The classes above are all parts of a model representing different types of vehicles and drivers, with fields and methods. That is the responsibility of these classes - to model these entities from real life.

Now imagine you needed to be able to store these objects in a database, and also serialize them to XML, JSON, or other formats. You want that implemented using a single method for each

operation, available on each `Car`, `Truck` or `Vehicle` object. A `store()` method, a `serializeToXML()` method and a `serializeToJSON()` method.

Please forget for a while, that implementing this functionality as methods directly on the objects may lead to a messy class hierarchy. Just imagine that this is how you want the operations implemented.

Where in the above diagram would you put these three methods, so they are accessible on all classes?

One way to solve this problem would be to create a common superclass for the `Vehicle` and `Driver` class, which has the storage and serialization methods. However, this would result in a conceptual mess. The class hierarchy would no longer model vehicles and drivers, but also be tied to the storage and serialization mechanisms used in your application.

A better solution would be to create some interfaces with the storage and serialization methods on, and let the classes implement these interfaces. Here are examples of such interfaces:

```
public interface Storable {

    public void store();
}
public interface Serializable {
    public void serializeToXML(Writer writer);
    public void serializeToJSON(Writer writer);
}
```

When each class implements these two interfaces and their methods, you can access the methods of these interfaces by casting the objects to instances of the interface types. You don't need to know exactly what class a given object is of, as long as you know what interface it implements. Here is an example:

Generic Interfaces

A generic Java interface is an interface which can be typed - meaning it can be specialized to work with a specific type (e.g. interface or class) when used. Let me first create a simple Java interface that contains a single method:

```
public interface MyProducer() {

    public Object produce();

}
```

This interface represents an interface which contains a single method called `produce()` which can produce a single object. Since the return value of `produce()` is `Object`, it can return any

Java object.

Here is a class that implements the `MyProducer` interface:

```
public class CarProducer implements MyProducer{
    public Object produce() {
        return new Car();
    }
}
```

The above class `CarProducer` implements the `MyProducer` interface. The implementation of the `produce()` method returns a new `Car` object every time it is called. Here is how it looks to use the `CarProducer` class:

```
MyProducer carProducer = new CarProducer();

Car car = (Car) carProducer.produce();
```

Notice how the object returned from the `carProducer.produce()` method call has to be cast to a `Car` instance, because the `produce()` method return type is `Object`. Using Java Generics you can *type* the `MyProducer` interface so you can specify what type of object it produces when you use it. Here is first a generic version of the `MyProducer` interface:

```
public interface MyProducer <T>{

    public T produce();

}
```

Now when I implement the `MyProducer` interface in the `CarProducer` class, I have to include the generic type declaration too, like this:

```
public class CarProducer<T> implements MyProducer<T>{

    @Override
    public T produce() {
        return (T) new Car();
    }
}
```

Now, when creating a `CarProducer` I can specify its generic interface type, like this:

```
MyProducer<Car> myCarProducer = new CarProducer<Car>();

Car produce = myCarProducer.produce();
```

As you can see, since the generic type for the `CarProducer` instance is set to `Car`, it is no longer necessary to cast the object returned from the `produce()` method, since the original method

declaration in the `MyProducer` interface states, that this method returns the same type as is specified in the generic type when used.

But - now it is actually possible to specify another generic type for a `CarProducer` instance than the type it actually returns from it's `produce()` method implementation. If you scroll up, you can see that the `CarProducer.produce()` implementation returns a `Car` object no matter what generic type you specify for it when you create it. So, the following declaration is possible, but would return in a `ClassCastException` when executed:

```
MyProducer<String> myStringProducer = new CarProducer<String>();

String produce1 = myStringProducer.produce();
```

Instead, you can lock down the generic type of the `MyProducer` interface already when you implement it, in the `CarProducer` class. Here is an example of specifying the generic type of a generic interface when implementing it:

```
public class CarProducer implements MyProducer<Car>{

    @Override
    public Car produce() {
        return new Car();
    }
}
```

Now you cannot specify the generic type of the `CarProducer` when using it. It is already typed to `Car`. Here is how using the `CarProducer` looks:

```
MyProducer<Car> myCarProducer = new CarProducer();

Car produce = myCarProducer.produce();
```

As you can see, it is still not necessary to cast the object returned by `produce()`, as the `CarProducer` implementation declares that to be a `Car` instance.

Reflection

Java Reflection - Classes

The Class Object

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. If you know the name of the class at compile time you can obtain a `Class` object like this:

```
Class myObjectClass = MyObject.class
```

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. For instance, if MyObject is located in package `com.jenkov.myapp` then the fully qualified class name is `com.jenkov.myapp.MyObject`

The `Class.forName()` method may throw a `ClassNotFoundException` if the class cannot be found on the classpath at runtime.

Class Name

From a `Class` object you can obtain its name in two versions. The fully qualified class name (including package name) is obtained using the `getName()` method like this:

```
Class aClass = ... //obtain Class object. See prev. section
String className = aClass.getName();
```

If you want the class name without the pacakge name you can obtain it using the `getSimpleName()` method, like this:

```
Class  aClass         = ... //obtain Class object. See prev. section
String simpleClassName = aClass.getSimpleName();
```

Modifiers

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. See prev. section
int modifiers = aClass.getModifiers();
```

Package Info

You can obtain information about the package from a `Class` object like this:

```
Class  aClass = ... //obtain Class object. See prev. section
Package package = aClass.getPackage();
```

From the `Package` object you have access to information about the package like its name. You can also access information specified for this package in the `Manifest` file of the JAR file this package is located in on the classpath. For instance, you can specify package version numbers in the `Manifest` file. You can read more about the `Package` class here: java.lang.Package

Superclass

From the `Class` object you can access the superclass of the class. Here is how:

```
Class superclass = aClass.getSuperclass();
```

The superclass class object is a `Class` object like any other, so you can continue doing class reflection on that too.

Implemented Interfaces

It is possible to get a list of the interfaces implemented by a given class. Here is how:

```
Class  aClass = ... //obtain Class object. See prev. section
Class[] interfaces = aClass.getInterfaces();
```

A class can implement many interfaces. Therefore an array of `Class` is returned. Interfaces are also represented by `Class` objects in Java Reflection.

NOTE: 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.

To get a complete list of the interfaces implemented by a given class you will have to consult both the class and its superclasses recursively.

Constructors

You can access the constructors of a class like this:

```
 Constructor[] constructors = aClass.getConstructors();
```

Constructors are covered in more detail in the text on Constructors.

Methods

You can access the methods of a class like this:

```
Method[] method = aClass.getMethods();
```

Methods are covered in more detail in the text on Methods.

Fields

You can access the fields (member variables) of a class like this:

```
Field[] method = aClass.getFields();
```

Fields are covered in more detail in the text on Fields.

Annotations

You can access the class annotations of a class like this:

```
Annotation[] annotations = aClass.getAnnotations();
```

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 Java `Constructor` object.

Obtaining Constructor Objects

The `Constructor` class is obtained from the `Class` object. Here is an example:

```
Class aClass = ...//obtain class object
Constructor[] constructors = aClass.getConstructors();
```

The `Constructor[]` array will have one `Constructor` instance for each public constructor declared in the 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 `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.

Constructor Parameters

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

```
Constructor constructor = ... // obtain constructor - see above
Class[] parameterTypes = constructor.getParameterTypes();
```

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.

Java 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`. This text will get into more detail about the Java `Field` object. Remember to check the JavaDoc from Sun out too.

Obtaining Field Objects

The `Field` class is obtained from the `Class` object. Here is an example:

```
Class aClass = ...//obtain class object
Field[] fields = aClass.getFields();
```

The `Field[]` array will have one `Field` instance for each public field declared in the class.

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.

Field Name

Once you have obtained a `Field` instance, you can get its field name using the `Field.getName()` method, like this:

```
Field field = ... //obtain field object
String fieldName = field.getName();
```

Field Type

You can determine the field type (String, int etc.) of a field using the `Field.getType()` method:

```
Field field = aClass.getField("someField");
Object fieldType = field.getType();
```

Getting and Setting Field Values

Once you have obtained a `Field` reference you can get and set its values using the `Field.get()` and `Field.set()` methods, like this:

```
Class  aClass = MyObject.class
Field field = aClass.getField("someField");

MyObject objectInstance = new MyObject();

Object value = field.get(objectInstance);

field.set(objetInstance, value);
```

The `objectInstance` parameter passed to the `get` and `set` method should be an instance of the class that owns the field. In the above example an instance of `MyObject` is used, because the

`someField` is an instance member of the `MyObject` class.

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 above.

Java 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`. This text will get into more detail about the Java `Method` object.

Obtaining Method Objects

The `Method` class is obtained from the `Class` object. Here is an example:

```
Class aClass = ...//obtain class object
Method[] methods = aClass.getMethods();
```

The `Method[]` array will have one `Method` instance for each public method declared in the class.

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);
```

Method Parameters and Return Types

You can read what parameters a given method takes like this:

```
Method method = ... // obtain method - see above
```

```
Class[] parameterTypes = method.getParameterTypes();
```

You can access the return type of a method like this:

```
Method method = ... // obtain method - see above
Class returnType = method.getReturnType();
```

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.

```
package hit.day16;

import java.util.Scanner;

public class ReflectionDemo {
    public static void main(String[] args)throws Exception {

        //1. static binding.... things which cannot be
changed..
        Paint staticPaint=new RedPaint();
        System.out.println("Object created through static
way...:"+staticPaint);

        //2. dynamic binding.... To dynamically create
objects...
        Scanner scan=new Scanner(System.in);
        System.out.println("Please enter the qualified
```

```
class name...:");
          String paintClass=scan.next();
          //Where ever the return type of a method is Object,
you need to do typecasting

          Paint
dynamicPaint=(Paint)Class.forName(paintClass).newInstance();
          System.out.println("Object created through dynamic
way...:"+dynamicPaint);

          //3. new way of creating objects
dynamically....after jdk9
          Paint
s=(Paint)Class.forName(paintClass).getConstructor().newInsta
nce();
          System.out.println("Object created through new
dynamic way...:"+dynamicPaint);


     }
}


abstract class Paint{

}
class RedPaint extends Paint{
     public RedPaint() {

     }
}
class BluePaint extends Paint{
     public BluePaint() {

     }
}
```

```
package hit.day16;
```

```
public class ReflectionDemo2 {
     public static void main(String[] args)throws Exception {
          Politician p=new AbcPolitician("ramu","father
ramu");
          System.out.println(p);

          //dynamic way...  old one...

     p=(Politician)Class.forName("hit.day16.AbcPolitician")
                    .getConstructor(new Class[]
{String.class,String.class}).newInstance("dynamicramu","dyna
mic father ramu");

          System.out.println("dynamic ramu...:"+p);
          //dynamic way .... new one  (they have achieved
this through VARARGS

     p=(Politician)Class.forName("hit.day16.AbcPolitician")

     .getConstructor(String.class,String.class).newInstance("
new dynamicramu","new father dynamic ramu");

          System.out.println("new dynamic ramu...:"+p);
     }
}

abstract class Politician {

}

class AbcPolitician extends Politician{
     String name;String fname;
     public AbcPolitician(String name,String fname) {
          this.name=name;this.fname=fname;
     }
     @Override
```

```
	public String toString() {
		return "The object is..:"+this.name+":"+this.fname;
	}
}
class XyzPolitician  extends Politician{

}
```

```
package hit.day16;

public class ReflectionDemo3 {
	public static void main(String[] args) throws Exception{

		//Students
s=(Students)Class.forName("hit.day16.JavaStudents").getConst
ructor().newInstance();

		Students
s=(Students)Class.forName("hit.day16.JavaStudents")
				.getDeclaredConstructor().newInstance();
		System.out.println(s);

		s=(Students)Class.forName("hit.day16.JavaStudents")

	.getDeclaredConstructor(String.class).newInstance("ramu"
);
		System.out.println(s);
	}
}
abstract class Students{

}
class JavaStudents extends Students{
	JavaStudents() {

	}
JavaStudents(String s) {
		System.out.println(s);
```

```
        }
}
```