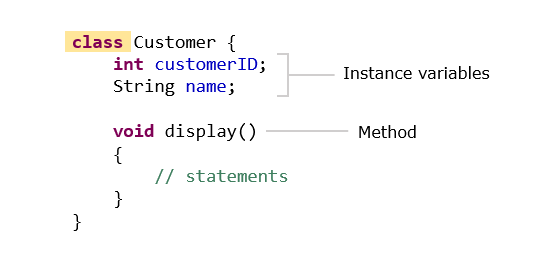
|  |
| --- |
| Java OOPS |
|  |

As we have seen, Java follows Object Oriented Programming concepts and classes and objects are basic concepts of Object Oriented Programming which revolve around the real-life entities.

A **class**is a design or blueprint that describes the characteristics and behaviors of a real-time entity. In other words, the class is a user-defined data type in Java.

* It starts with the keyword "class" followed by a name.
* It specifies attributes (characteristics) and methods (behaviors).
* Attributes are the elements (instance variables) which hold the values of a particular entity that define its characteristics.
* Methods are the sets of instructions that define the behaviors of the entity.

An example of creating a class can be seen below,



In the above class definition, we can see that the attributes are the variables of the class. But what is a method and what is its importance in a class?

So, let us discuss about methods next.

A **method**is a set of statements which depicts the behavior of a class. We have already encountered the main method.

To execute any program, the compiler always needs an initial/entry point to start from. In case of Java programs, that initial/entry point is marked by the main()method present in the program. Without this method, the JVM cannot start executing the program. The syntax can be given as:

1. public static void main(String[] args){
2. *//statements*
3. }

**public:** This specifies the level of access of the method within and outside the program. And to make main() method visible to JVM, the level of access must be set to "public". If the same is anything except public (private, protected or default) the JVM will be unable to see the initial/entry point of the program. And hence the program will not run.

**static:** Generally, a Java program is in the form of a class. And whenever we create classes in Java, we need to create instances/objects to access the class methods and attributes. The same applies to main() method also. But by declaring a class method as "static", we can call tha method without creating instances of the class. Hence we declare the main() method as static.

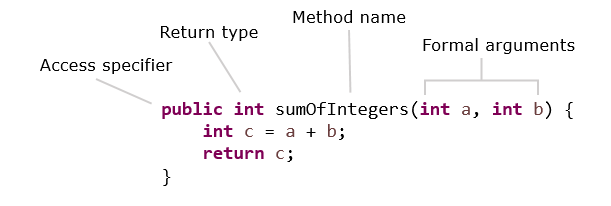
**void:**In Java, every method is expected to return a value of any type (integer, double, string etc.). And to make a method not return any value, it must be declared "void". Since the main() method is not expected to return any value, it must be declared as void.

**main():**As discussed above, a Java program is basically a class. And the main() method is the default method of this class. This method marks the initial/entry point of the Java program.

**String args[]:**Now a program in general may take inputs from the users. In Java, they are usually in the form of multiple strings as "command line arguments" grouped as an array of string. This string array is taken as the argument in the main() method.

Now, let us define a method.

A method definition looks as shown below:



The first line of the method contains the method signature, which includes the name of the method and the arguments (parameters) it takes.

Methods need to be called or invoked to execute the statements present inside the method.

The code given below demonstrates a method and the way to call or invoke the method.

1. public class Calculator {
2. *// Method definition - here, numberOne and numberTwo are formal parameters or arguments*
3. int calculateSum(int numberOne, int numberTwo) {
4. return numberOne+numberTwo;
5. }
6. }
7. public class Tester {
8. public static void main(String[] args) {
9. Calculator calculator=new Calculator();
11. int numberOne=11;
12. int numberTwo=1;
14. *//Method call - here, numberOne and numberTwo are actual parameters or arguments*
15. int sum=calculator.calculateSum(numberOne, numberTwo);
16. System.out.println("The sum is "+sum);
17. }
18. }

Arguments of a method are optional. The arguments specified in the method definition are called formal arguments or parameters. The arguments or parameters that are used in the method call are known as actual arguments or parameters.

Methods may or may not return a value after execution.

* The return keyword is used to return a value from the called method to the calling method.
* The return type specifies the data type of the value to be returned.
* If a method does not return anything, the return type should be specified as void.

In the code given above, the value returned by the method will be stored in the variable sum.

Next, let us see how to create objects.

Till now we have learnt to create a class. A Java class, is a blueprint so how do we use it? It can be done by providing some values to the attributes of the class. This is called the instantiation of a class and the entity we get after instantiation is called the object.

An object is an instance of a class.

* It allows us to use the attributes and behaviors specified in the class.
* A class can have any number of objects.
* An object holds data related to one instance of a class.
* In Java, an object is created by using the new keyword.

Let's create an object for the Customer class shown below:

1. class Customer {
3. String customerName;
4. *// getter method - returns the customerName of the object*
5. String getCustomerName() {
6. return customerName;
7. }
9. *//setter method - sets the customerName of object to the specified value*
10. void setCustomerName(String customerName) {
11. this.customerName = customerName;
12. }
13. }

An object for the class can be created as shown below:

1. Customer customerObject = new Customer();

Methods and attributes can be invoked by using the "." operator:

1. customerObject.customerName = "Ron";
2. System.out.println("Hello "+customerObject.getCustomerName()); *//Output : Hello Ron*
4. *// To change the name we use setter method*
5. customerObject.setCustomerName("Jack");
6. System.out.println("Hello "+customerObject.getCustomerName()); *//Output : Hello Jack*

**Note :**

* During object creation, the instance variables are automatically assigned default values (for example: 0 for integer, 0.0 for double, null for String etc.) based on their respective datatypes.
* Local variables declared inside methods have to be initialized with relevant values before they can be used.

Looking at the object creation, do you notice that we are calling a function having the same name as that of our class? We have created only one method in the Customer class, i.e., display(). There is no function with the name Customer(). So how are we calling it?

Constructor and this keyword:

A constructor is a method which has the same name as the class. It is a special method as it does not return any value, and is used for creating objects and initializing their values.

Syntax:

1. <<classname>>(){  }

For Example:

1. *//Constructor*
2. Customer(){ }
3. *//Constructor above is called while creating objects as shown below,*
4. Customer custObj = new Customer();

Here, Customer() indicates constructor invocation during object creation

**Types of constructors:**

* Non-parameterised/Default constructor

1. A constructor which doesn't take any parameter
2. If the programmer doesn't provide any constructor, a non-parameterized/default constructor will be provided by the compiler.

* Parameterised Constructor

1. A constructor which accepts parameters (arguments)
2. *//Parameterised constructor*
3. Customer(int customerId) {
4. System.out.println(customerId);
5. }
6. *//Constructor above is called while creating objects as shown below,*
7. Customer custObj = new Customer(1001);

Note: If the programmer provides a parameterized constructor then the default constructor is not provided by the compiler.

Next, let us learn about the 'this' keyword.

Now let us consider a situation where a method has an argument or a local variable which has its name similar to an instance variable. Who gets the higher preference, the local variable/argument or the instance variable?

Surprisingly, the local variable gets the higher preference than the instance variable inside the method. And this is known as shadowing of a field. So, how do we use both the instance variable and the local variable/argument together?

How do we call any constructors or methods of a class, from a method of the class?

1. class Customer{
2. String customerName;
3. public Customer(String customerName) {
4. customerName = customerName;
5. }
6. public void display () {
7. System.out.println("Customer Name : "+customerName); *// Output :- Customer Name : null*
8. }
9. }
10. public class Tester {
11. public static void main(String[] args) {
12. Customer customer1 = new Customer("Jack");
13. customer1.display();
14. }
15. }

Here, the variable 'customerName' used in the constructor is local to that constructor. The instance variable (customerName) name will remain null.  
To assign the value passed into the constructor to the instance variable 'customerName' and to call the display() method, this keyword can be used as shown below:

1. class Customer{
2. String customerName;
3. public Customer(String customerName) {
4. this.customerName = customerName;
5. }
6. public void display () {
7. System.out.println("Customer Name : "+customerName); *// Output :- Customer Name : Jack*
8. }
9. }

As we have learnt to invoke attributes using this, can we invoke methods using this? Let us see how to do that next.

this keyword is also used to invoke the constructor or any method of the current object.

Observe the code given below:

1. public class Customer {
2. public String customerId;
3. public String customerName;
4. public long contactNumber;
5. public String address;
6. public Customer() {
7. System.out.println("Parameterless constructor called");
8. }
9. public Customer(String customerId, String customerName, long contactNumber,
10. String address) {
11. *// this() is used to invoke the constructor of the current class*
12. *// Since no parameters are specified, parameterless constructor will be invoked*
13. this();
14. this.customerId = customerId;
15. this.customerName = customerName;
16. this.contactNumber = contactNumber;
17. this.address = address;
18. }
19. public void displayCustomerName() {
20. System.out.println("Customer Name : " + customerName);
21. }
22. public void displayCustomerDetails() {
23. System.out.println("Displaying customer details \n\*\*\*\*\*\*\*\*\*\*\*");
24. System.out.println("Customer Id : " + customerId);
26. *//this is used to invoke method*
27. this.displayCustomerName();
28. System.out.println("Contact Number : " + contactNumber);
29. System.out.println("Address : " + address);
30. }
31. }

We will create a Tester class for the same, which will have input as follows:

1. public class Tester {
2. public static void main(String[] args) {
3. Customer customer = new Customer();
4. customer.displayCustomerName();
5. customer.displayCustomerDetails();
7. Customer customer1 = new Customer("K123","Katy",7865l,"21A, Downtown, LA");
8. customer1.displayCustomerName();
9. customer1.displayCustomerDetails();
10. }
11. }

For an object customer, we are calling parameterless constructor which is printing the statement "Parameterless constructor called" and is not setting any attributes, hence it will set them to null. And customer1 is calling the parameterized constructor which is calling parameterless constructor (using this() ) and setting the value of the attributes.

After executing the above code we will get the output as :

1. Parameterless constructor called
2. Customer Name : null
3. Displaying customer details
4. \*\*\*\*\*\*\*\*\*\*\*
5. Customer Id : null
6. Customer Name : null
7. Contact Number : 0
8. Address : null
9. Parameterless constructor called
10. Customer Name : Katy
11. Displaying customer details
12. \*\*\*\*\*\*\*\*\*\*\*
13. Customer Id : K123
14. Customer Name : Katy
15. Contact Number : 7865
16. Address : 21A, Downtown, LA

Class Diagram:

By now you learnt how to create a class, its attributes or instance variables, methods and constructors. It is also important to understand how to represent them so that anyone can understand about the class even without comprehensive knowledge in programming languages. Class diagrams helps in achieving this.

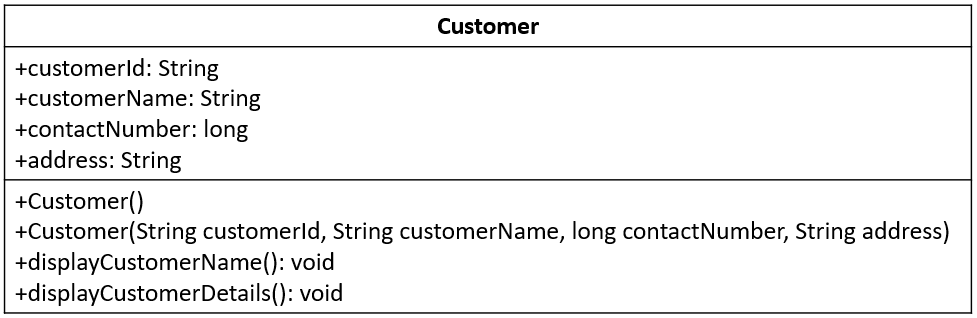
Class diagrams are used for diagrammatically representing the classes, its structure and relationship with other class.

Class diagram mainly describes the attributes and methods of a class. Relationships and the dependencies between Java classes, interfaces, enums, fields, methods, references etc. can also be modelled using a class diagram.

It also helps in constructing executable code of an enterprise application. Class diagram is the only UML (Unified Modeling Language) diagram which can be directly mapped to object-oriented languages and therefore, is one of the most widely used UML diagrams.

In UML class diagrams, a class is represented by a rectangular box with its class name on top and divided into different sections. The first section represents all the attributes of the class and the second represents methods.

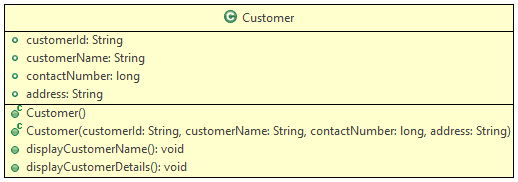
Below is the UML class diagram of a class Customer which we discussed in the previous tryout.



The symbol + denotes that the members of the class are publicly accessible. Similarly, there are other symbols representing accessibility of members.

Class diagrams can also be generated using IDEs.

Below is the class diagram for the class Customer, generated using Eclipse IDE.



Each symbol has its own meaning as per the access modifier and the type of attributes and methods.

The green color represents the class or its variables are having public accessibility.

Coding Standards:

Now that we have seen how we can work with classes and objects, let us discuss few coding standards that are followed in Java.

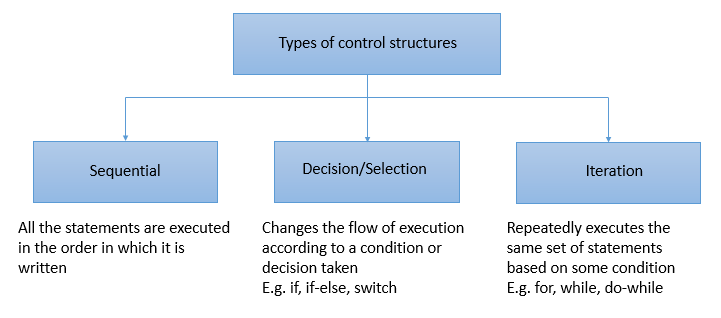
Java follows certain conventions when it comes to naming variables, packages, classes, methods, and interfaces i.e. **identifiers**. The programs are more understandable if meaningful names are provided for the identifiers. The following conventions are followed:

* **Identifiers**can contain alphabets, numbers, underscore( \_ ), and the dollar sign. ( $ ). There is no restriction in the length.
* Reserved keywords **should not** be used for naming identifiers.
* **Class** names should follow PascalCasing i.e the first letter and the first letter of each internal word should be capitalized. Example: StringBuffer, EmployeeDemo
* **Method** names should follow camelCasing i.e the first letter should be in lowercase and the first letter of each internal word should be in uppercase. Example: display, getEmployeeName
* **Variable**names should start with lowercase and the first letter of the subsequent words should be in uppercase. Example: total, averageCount
* **Constants**should be in uppercase with each word separated by an underscore. Example: PI, MIN\_LENGTH

Let us discuss few other concepts such as control structures next.

Control Structures:

In a program, the instructions are usually executed line by line. Sometimes, all the statements in a program may not be executed. There can be a change in the flow of control and can be implemented using control structures.



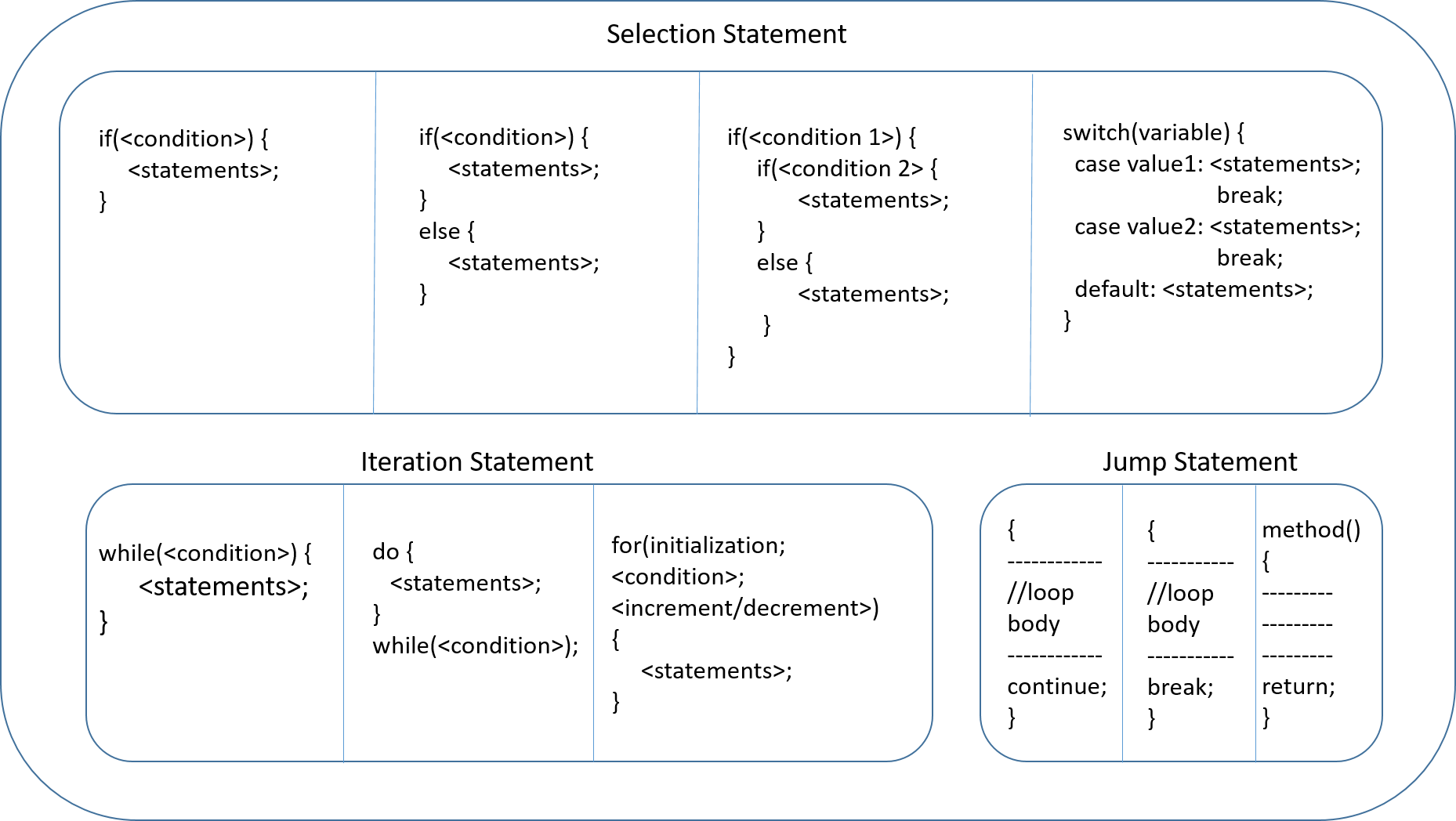
The selection control structures available in Java are:

* if-else
* if-else if
* switch

The iteration control structures available in Java are:

* while loop
* do-while loop
* for loop
* for-each loop

The syntax of the various control structures is given below.



Introduction To Package:

Imagine walking into a library to get your favorite novel, but welcomed by a room full of an unorganized stack of books. Would you still go ahead and search for the novel?

Of course not!

Just as books are cataloged and arranged in a library, it is important for us to organize our Java files. This can be done with the help of packages.

A **package**is a set of logically related classes.

A package helps in controlling access and avoiding name conflicts among them.

You need to provide unique names to packages inside a project.

**How would you name a package?**

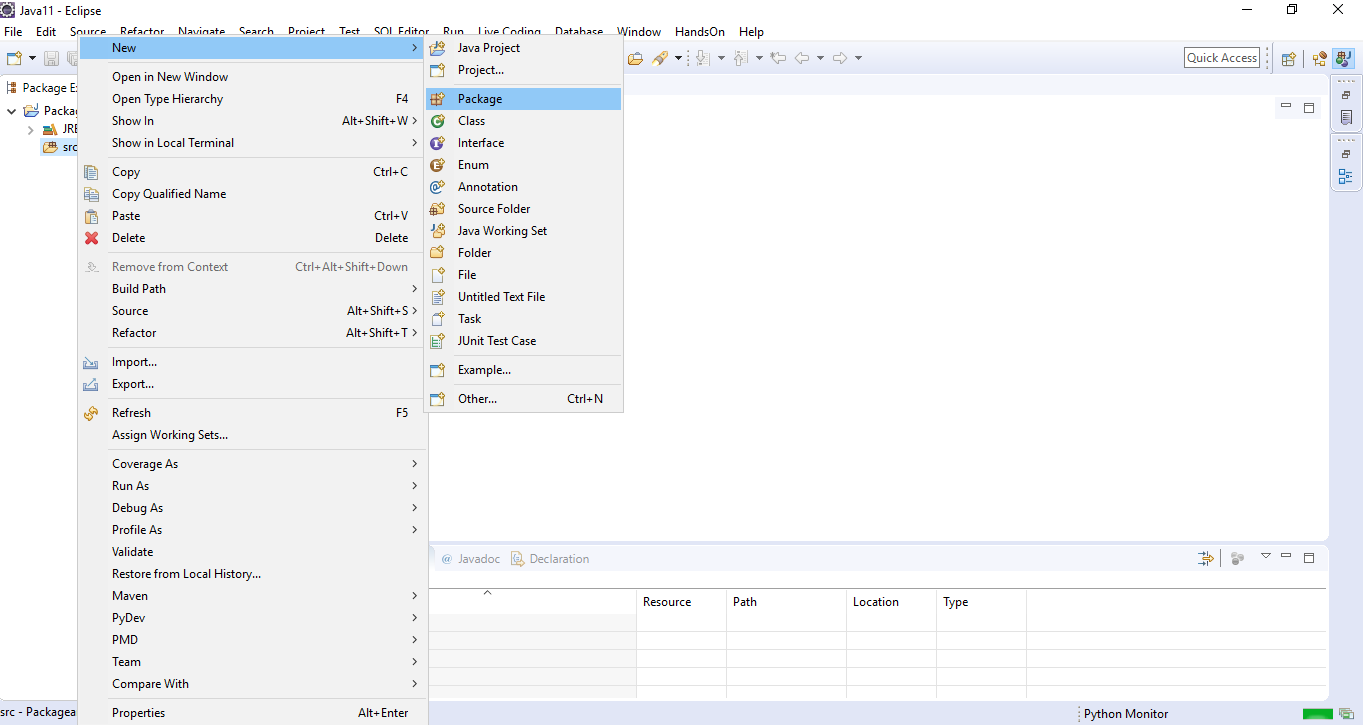
The convention for naming packages is to use all lower case alphabets. This is done to avoid conflicts with names of classes.

Also, organizations use reverse internet domains for naming packages.

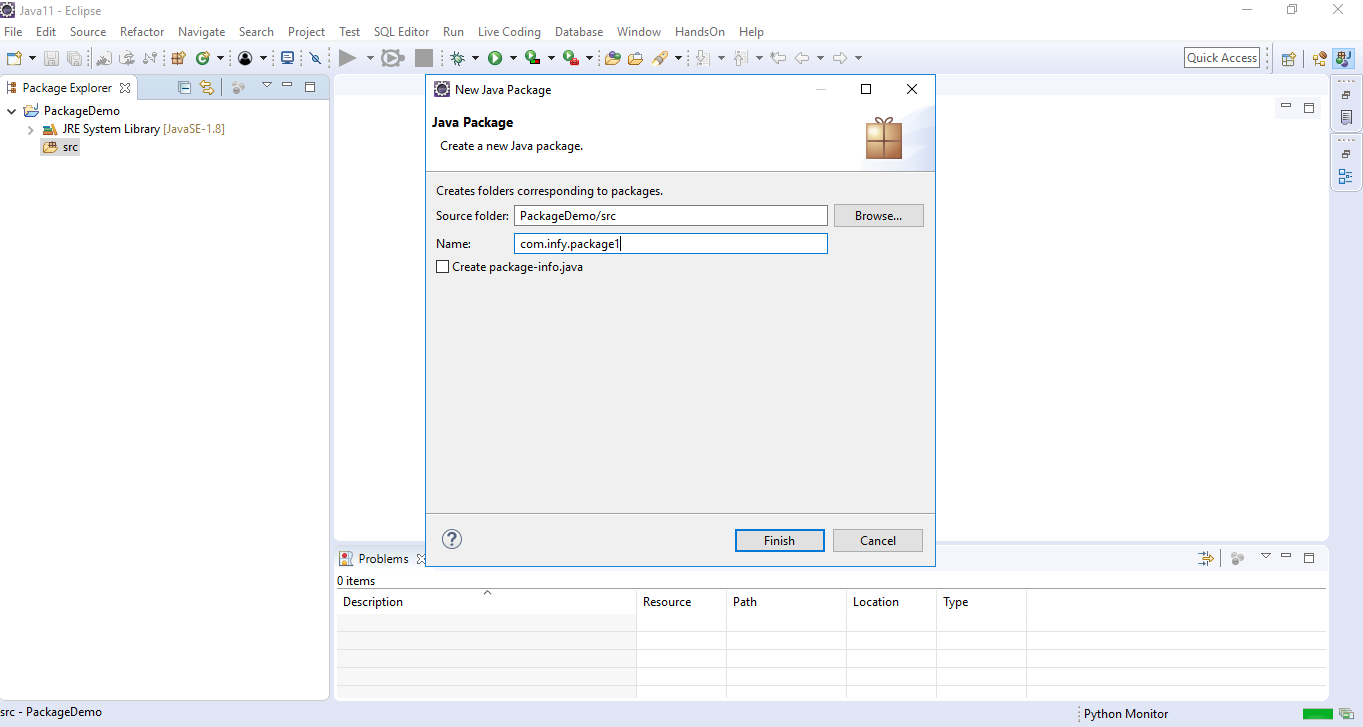
E.g. -  A package created for an internal application of Infosys can be named as com.infy.

Now that you know about the package and its naming convention, you will now see how to create a package.

To **create a package**, right-click on src and select New>Package to create a new package as shown below.

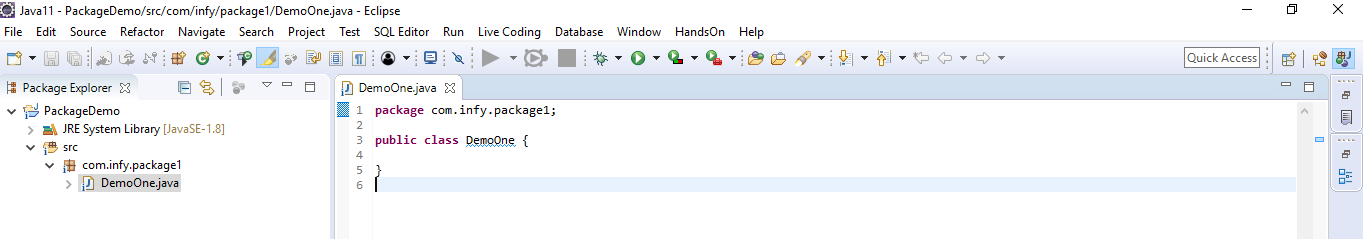


Next, you need to provide a name to the package as shown below.



The package is named as 'com.infy.package1' in the screenshot given above. Please observe that each alphabet is in lower case and also reverse internet domain has been used while naming the package.

Once a class is created inside a package, the package keyword specifies the name of the package as shown below. This should be the first line of a .java file.



Classes of one package can also be used in other packages.

**How do you use classes of one package in another package?**

To use a class of one package in another package, an **import**statement needs to be used which will be demonstrated to you through a demonstration shown later.

Let us discuss access modifiers next.

Access Modifiers:

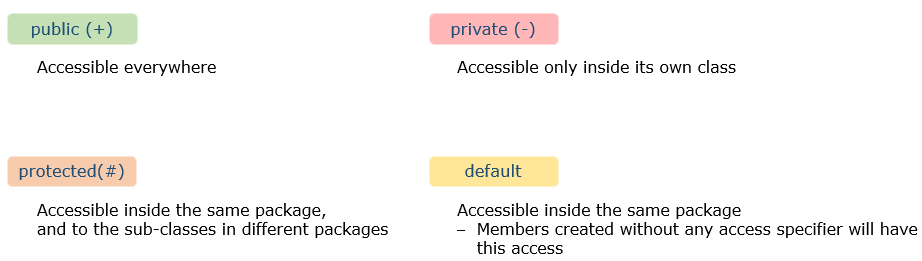
Now that you have understood packages, let us also discuss access modifiers.

You would have seen the keyword 'public' in multiple places. The keyword public is an access modifier.

Access modifiers are used to control the visibility of a class and its members. This facilitates encapsulation.

There are 4 access modifiers in Java  - public, private, default and protected.

Out of these 4, you will learn more about protected later in th e course.



**Note**: A class can have only public or default access.

The visibility of members across classes and packages are shown below.

​​

| **Visibility of Access Modifiers** | | | | |
| --- | --- | --- | --- | --- |
| **Members accessible to** | **public** | **protected** | **deafult** | **private** |
| **Same class** | **yes** | **yes** | **yes** | **yes** |
| **All classes in same package** | **yes** | **yes** | **yes** | **no** |
| **Sub-classes in different packages** | **yes** | **yes** | **no** | **no** |
| **All classes in different packages** | **yes** | **no** | **no** | **no** |

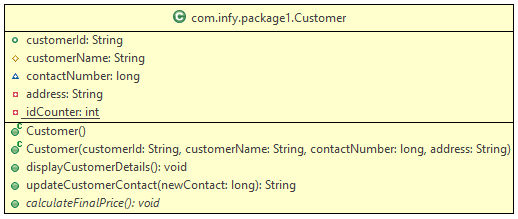
You will now see a demonstration on packages and access modifiers to understand them better.

Since you have now learnt about the different access modifiers, we will see how they are represented in class diagrams.

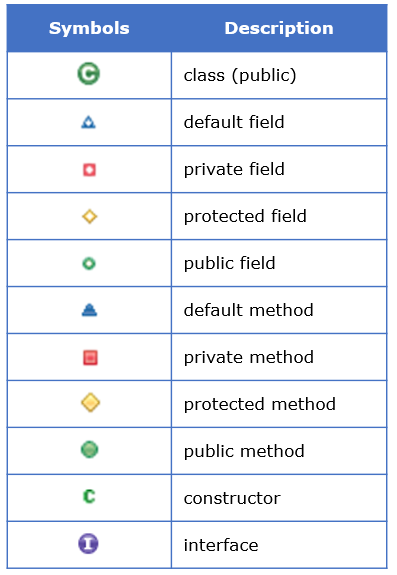
Consider the below code snippet for the class Customer.

1. public abstract class Customer {
3. public String customerId;
4. protected String customerName;
5. long contactNumber;
6. private String address;
8. private static int idCounter;
9. public Customer() {
10. System.out.println("Parameterless constructor called");
11. }
12. public Customer(String customerId, String customerName, long contactNumber, String address) {
13. this.customerId = customerId;
14. this.customerName = customerName;
15. this.contactNumber = contactNumber;
16. this.address = address;
17. }
18. public void displayCustomerDetails() {
19. System.out.println("Displaying customer details \n\*\*\*\*\*\*\*\*\*\*\*");
20. System.out.println("Customer Id : " + this.customerId);
21. System.out.println("Customer Name : " + this.customerName);
22. System.out.println("Contact Number : " + this.contactNumber);
23. System.out.println("Address : " + this.address);
24. }
26. public String updateCustomerContact(long newContact) {
27. String message = null;
28. if (this.contactNumber != newContact) {
29. this.contactNumber = newContact;
30. message= "Contact updated";
31. }
32. else
33. message = "Provide a new contact";
35. return message;
37. }
39. public abstract void calculateFinalPrice();
40. }

Below is the class diagram for the above given Customer class.



Different symbols used in the above class diagram is explained below:

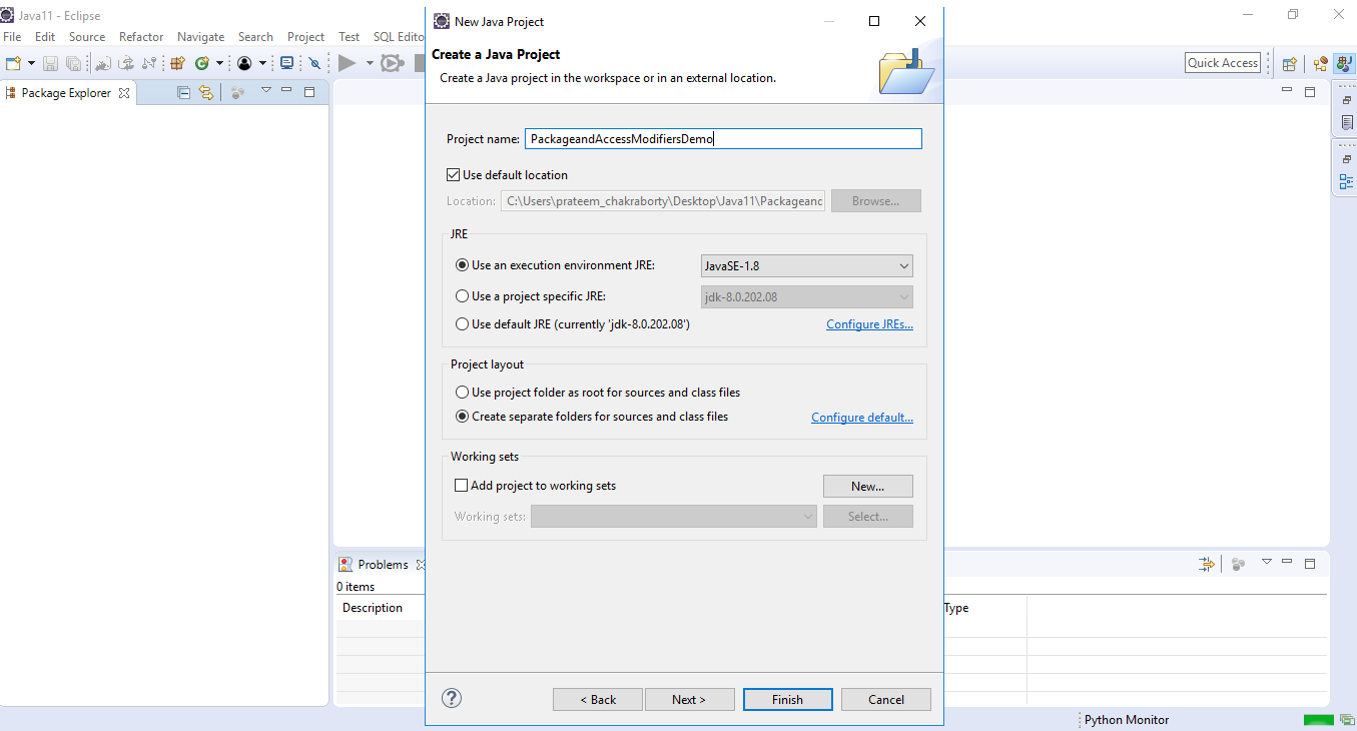


Also static variables and methods are represented using an underline given to them and abstract methods are represented in italics.

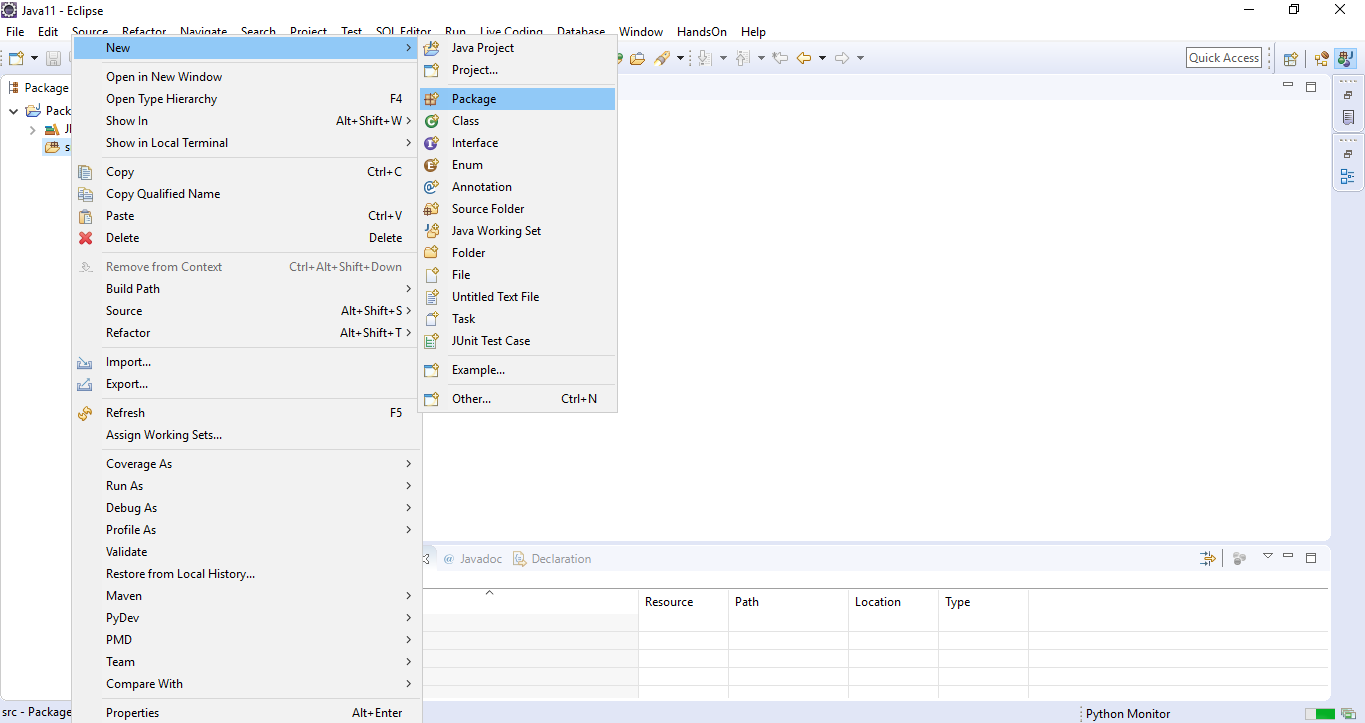
Static, Abstract and Interface concepts will be explained later in the course.

Please follow the steps given below to understand packages and public, private and default access modifiers in Java.

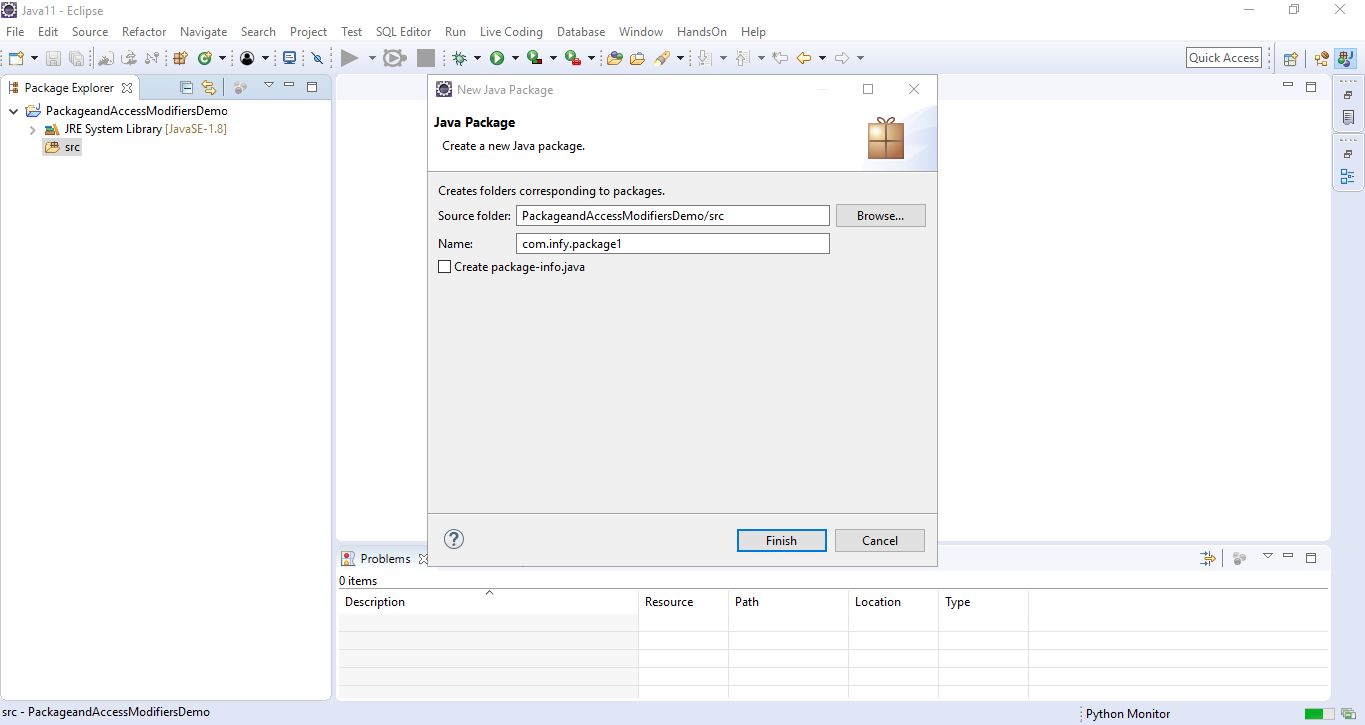
**Step 1**: Create a project named PackageandAccessModifiersDemo as shown below.



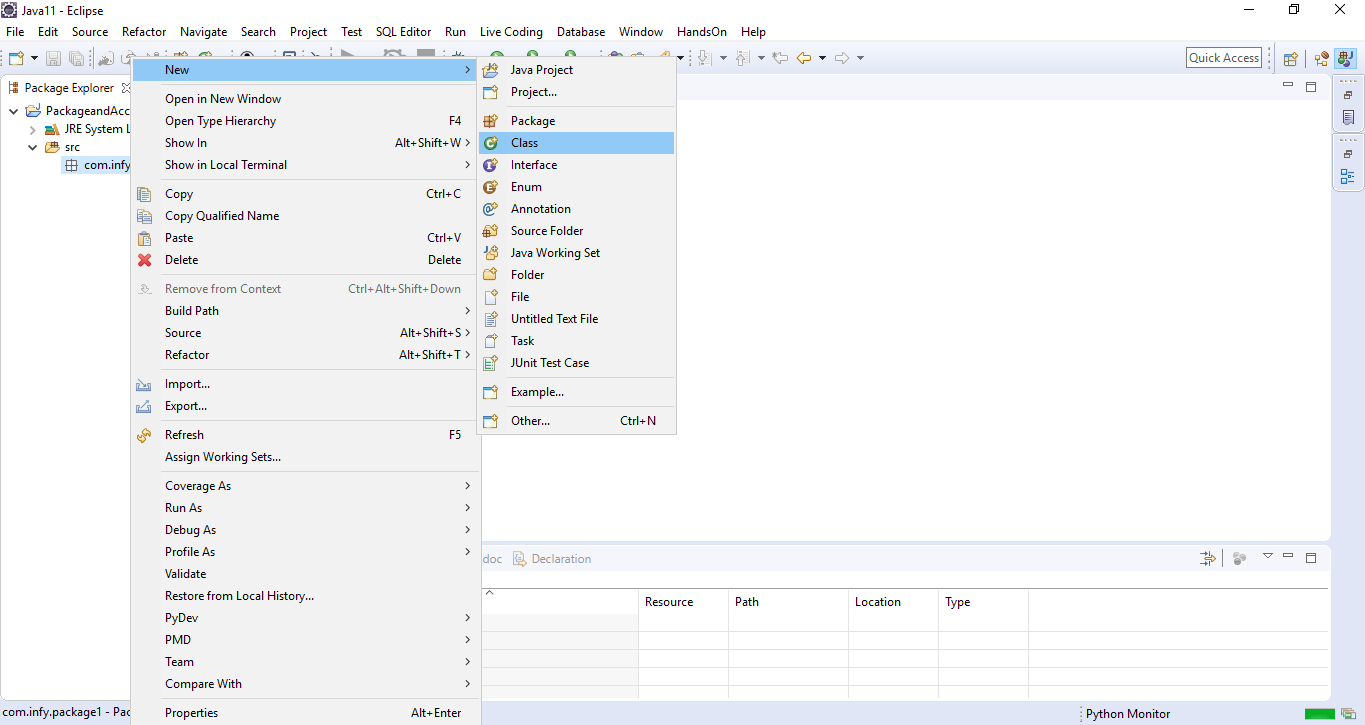
**Step 2**: Right click on src and select New>Package to create a new package as shown below.



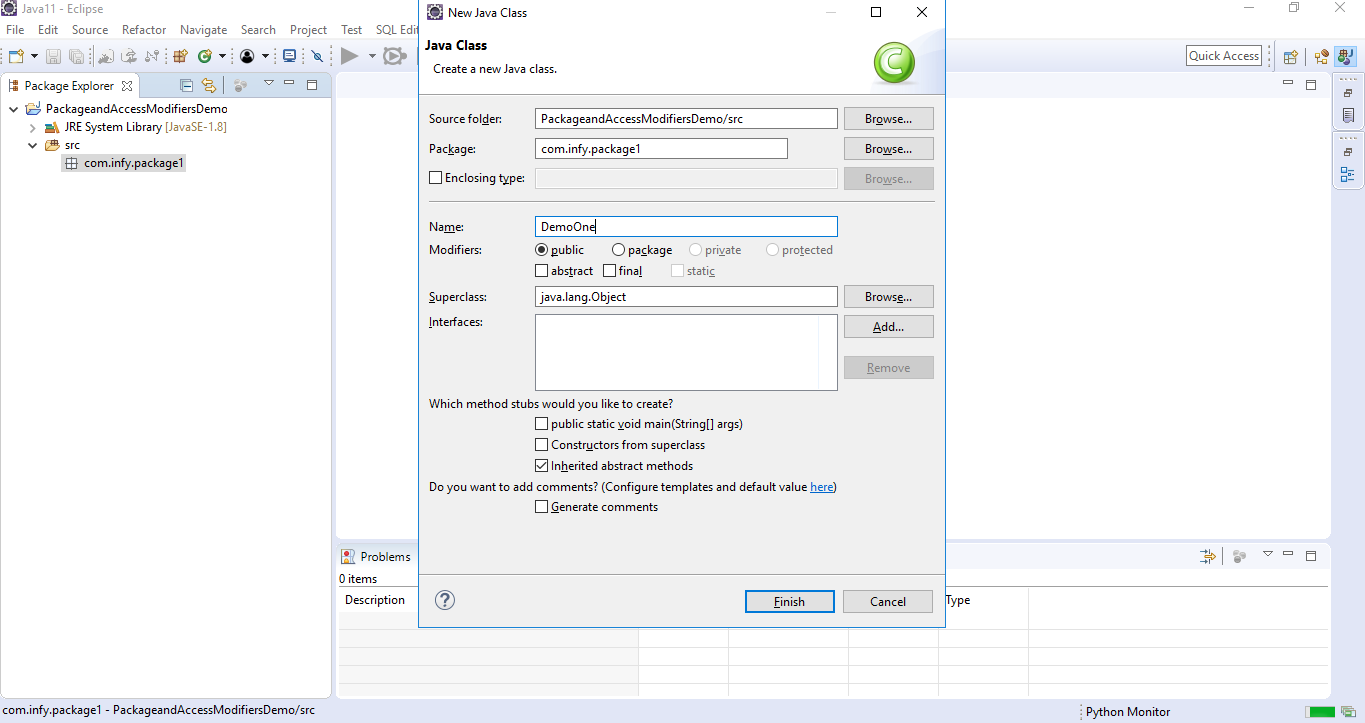
Name the package as com.infy.package1.



**Step 3**: Right click on the package name and select New>Class to create a new class in com.infy.package1 as shown below.



Name the class as DemoOne.



**Step 4**: Create three variables named variableOne, variableTwo and variableThree with public, private and default access modifiers respectively in the class DemoOne.

1. public class DemoOne {
2. public int variableOne;
3. private int variableTwo;
4. int variableThree;
5. }

**Step 5**: Add another class named DemoTwo in com.infy.package1.

**Step 6**: Create a method checkAccessibility as shown below in the class DemoTwo. This method will be used to check the accessibility of the variables of class DemoOne in class DemoTwo.

1. public class DemoTwo {
2. public void checkAccessibility() {
3. }
4. }

**Step 7**: Create an object of class DemoOne in the method checkAccessibility as shown below.

1. public class DemoTwo {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. }
5. }

**Step 8**: Try to access and initialize variableOne in the method checkAccessibility() as shown below.

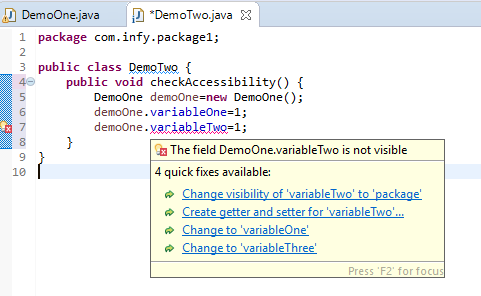
1. public class DemoTwo {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=1;
5. }
6. }

You can observe that variableOne of class DemoOne is accessible in the class DemoTwo as variableOne is public and hence, can be accessed anywhere.

**Step 9**: Now, try to access variableTwo in the method checkAccessibility() as shown below.

1. public class DemoTwo {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=1;
5. demoOne.variableTwo=1;
6. }
7. }

You can observe that you get a compilation error stating 'The field DemoOne.variableTwo is not visible' while trying to access variableTwo of class DemoOne in the class DemoTwo as variableTwo is private and hence, can be accessed only within the class in which it is defined.



**Step 10**: Now, try to access variableThree and initialize it in the method checkAccessibility() as shown below.

1. public class DemoTwo {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=1;
5. demoOne.variableThree=1;
6. }
7. }

You can observe that variableThree of class DemoOne is accessible in the class DemoTwo as variableThree is default and hence, can be accessible anywhere within the same package.

**Step 11**: Add another package named com.infy.package2 in the same project.

**Step 12**: Create a class named DemoThree in com.infy.package2.

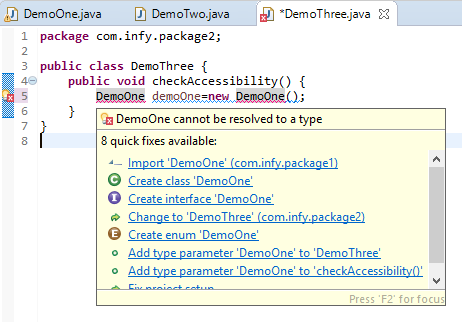
**Step 13**: Create a method checkAccessibility as shown below in the class DemoThree. This method will be used to check the accessibility of the variables created in class DemoOne in class DemoThree.

1. public class DemoThree {
2. public void checkAccessibility() {
3. }
4. }

**Step 14**: Create an object of class DemoOne in the method checkAccessibility as shown below.

1. public class DemoThree {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. }
5. }

You will now observe a compilation error stating 'DemoOne cannot be resolved to a type' as shown below.



This error is coming as the classes DemoOne and DemoThree are present in different packages.

**Step 15**: To use or access the class DemoOne in the class DemoThree, you can use com.infy.package1.DemoOne, i.e., <package\_name>.<class\_name> as shown below.

1. public class DemoThree {
2. public void checkAccessibility() {
3. com.infy.package1.DemoOne demoOne=new com.infy.package1.DemoOne();
4. }
5. }

In this approach, you have to use the fully qualified name (<package\_name>.<class\_name>) every time you need to access the class. Hence, this is not a recommended approach.

Instead, it is recommended to import the required classes using **import**statement.

**Step 16**: Import the class named DemoOne from com.infy.package1 to com.infy.package2 using the import statement as shown below.

1. package com.infy.package2;
2. import com.infy.package1.DemoOne;
3. public class DemoThree {
4. public void checkAccessibility() {
5. DemoOne demoOne=new DemoOne();
6. }
7. }

To import all the classes present inside a given package, import <package\_name>.\* can be used.

If you use <package\_name>.\*, then all the classes of the package will be imported but the classes present inside the sub packages will not be available for use.

**Step 17**: Now that you have imported  class DemoOne in com.infy.package2, try to access variableOne and initialize it in the method checkAccessibility() as shown below.

1. public class DemoThree {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=1;
5. }
6. }

You can observe that variableOne of class DemoOne is accessible in the class DemoThree as variableOne is public and hence, can be accessed anywhere.

**Step 18**: Now, try to access variableTwo in the method checkAccessibility() as shown below.

1. public class DemoThree {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=1;
5. demoOne.variableTwo=1;
6. }
7. }

You can observe that you get a compilation error stating 'The field DemoOne.variableTwo is not visible' while trying to access variableTwo of class DemoOne in the class DemoThree as variableTwo is private and hence, can be accessed only within the class in which it is defined.

**Step 19**: Now, try to access variableThree and initialize it in the method checkAccessibility() as shown below.

1. public class DemoThree {
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=1;
5. demoOne.variableThree=1;
6. }
7. }

You can observe that you again get a compilation error stating 'The field DemoOne.variableTwo is not visible' while trying to access variableThree of class DemoOne in the class DemoThree as variableThree is default and hence, can be accessed only within the package in which it is defined.

So, through this demo, you would have understood the scope of the different access modifiers and how to access a class of one package in another using import statement.

String Introduction:

Next, we will look at some of the built-in classes available in Java.

The first built-in class that we will see is the String class.

As we all know already, a sequence of characters can be stored using a variable of String.

There are two ways to create a string:

**String literal**

E.g. String customerName = "Jasmine";

**Using new() keyword**

E.g. String customerName = new String("Jasmine");

In the following demo, we will understand how to create string in two ways.

1. *//String creation using = operator*
2. String name = "Jasmine";
3. *//String creation using new()*
4. String customerName = new String("Jasmine");
5. System.out.println(name); *//Output: Jasmine*
6. System.out.println(customerName); *//Output: Jasmine*

# **String methods**

Java String class provides a collection of built-in methods which help in performing various operations on string. Some of which are:

| String methods | |
| --- | --- |
| Method | ****Description**** |
| **String concat(String str)** | **Concatenates the specified string with this string** |
| **int length()** | **returns the number of characters in a string** |
| **boolean equals(String str)** | **checks case sensitive equality of the string** |
| **boolean equalsIgnoreCase(String str)** | **checks case insensitive equality of the string** |
| **String replace(char old, char new)** | **replaces all the occurence of specified character with new character** |
| **int compareTo(String anotherString)** | **compares two strings in lexicographical order and returns an int value** |
| **String toUpperCase()** | **converts all of the characters in this String to upper case** |
| **String toLowerCase()** | **converts all of the characters in this String to lower case** |
| **boolean isEmpty()** | **returns true if the String is empty or length() is 0** |
| **boolean startsWith(String prefix)** | **checks whether string starts with the specified prefix** |
| **boolean endsWith(String suffix)** | **checks whether string ends with the specified suffix** |
| **String substring(int beginIndex, int endIndex)** | **returns a substring of the string that begins at the specified beginIndex and ends at the character one less than endIndex** |
| **String[ ] split(String regex)** | **splits the string around matches of the given regular expression and returns an array** |
| **int indexOf(String str)** | **returns the index within the string of the first occurence of the specified substring** |
| **String trim()** | **returns a string with all leading and trailing space removed** |

Few of the methods that are introduced since Java 11 are:

| New string methods introduced since Java 11 | |
| --- | --- |
| Method | ****Description**** |
| **String repeat(int count)** | **repeats the string having count passed as parameter** |
| **boolean isBlank()** | **checks if the string is empty or contains only white spaces** |
| **String strip()** | **returns a string with all the leading and trailing white spaces removed** |
| **String stripLeading()** | **returns a string with all the leading white spaces removed** |
| **String stripTrailing()** | **returns a string with all the trailing white spaces removed** |
| **Stream<String> lines()** | **splits the string by its line terminators and returns a Stream of Strings** |

All these methods will be discussed in the tryout.

In the demo given below, we compare two strings using equals and equalsIgnoreCase method. We manipulate the given string using methods like replace() and concat(). Similarly, using other methods we can manipulate the strings.

1. String name = "Thomas";
2. String customerName = new String("THOMAS");
3. boolean result1 = name.equals(customerName); *// result1 = false*
4. boolean result2 = name.equalsIgnoreCase(customerName); *//result2 = true*
6. String username = name.concat("#24"); *// will concatenate #24 to name i.e Thomas#24*
7. String newString = customerName.replace("M", "m"); *// Will replace all occurence of M to m*
8. *// i.e THOmAS*

In the next page we will discuss the immutable nature of string and will figure out an alternative to String.

String Immutability and StringBuilder:

Consider a scenario, where we have to modify the variable name = "Oliver" to "Oliver Carter". To implement this we will use concat method as shown below.

1. String name = "Oliver";
2. name.concat(" Carter");
3. System.out.println(name); *//Output Oliver*

In the above example, we have string "Oliver" referred by the variable name. Even after using concat, name("Oliver") is not changed, rather a new object is created "Oliver Carter", having no reference. This is because String in Java is immutable. Immutable means which cannot be changed.

To solve this we will use a mutable class known as StringBuilder. StringBuilder class consists of the append method that concatenates the string with the specified argument which might solve our problem. Let's learn more about StringBuilder Class.

# StringBuilder Class

StringBuilder Objects are similar to String objects but they can be modified hence they are mutable. This means if we try to modify the original string, a new string will not be generated instead original string will get changed. In one way we can say that StringBuilder will reduce memory usage. It provides an alternative to the String class, it is mutable and easy to use.

The creation of StringBuilder object is shown below

1. *//create StringBuilder object*
2. StringBuilder name = new StringBuilder();
3. name.append("Oliver");
4. name.append(" Carter");
5. System.out.println(name); *//Output: Oliver Carter*

As given in the code, we can modify the object using the append method and can create StringBuilder objects in a different way. We will discuss some of the methods of StringBuilder.

Some of the most used methods of StringBuilder are given as follows:-

| StringBuilder methods | |
| --- | --- |
| Method | ****Description**** |
| **StringBuilder append(String str)** | **appends the specified string with the string** |
| **StringBuilder insert(int offset, String str)** | **inserts the given string at the specified position** |
| **StringBuilder reverse()** | **reverses the given string** |
| **char charAt(int index)** | **returns the character present at the specified position** |
| **StringBuilder delete(int start, int end)** | **deletes the string starting from start to one less than end** |

The implementation of these methods will see in the demo given below.

1. *//creation of StringBuilder Object with capacity 50.*
2. StringBuilder name = new StringBuilder(50);
4. name.append("Mississippi");
5. int length = name.length(); *// will give the length of address*
6. name.insert(length, " River");
7. System.out.println(name); *//Output :- Mississippi River*
9. name.reverse(); *// Output :- reviR ippississiM*
10. name.delete(5, 10); *// Output :- reviRssissiM*
11. System.out.println(name.charAt(3)); *// Output :- i*

We will explore some more methods of StringBuilder in the tryout.

Wrapper Class In Java:

All this time, we have been using primitive types. But there are occasions where we need to represent them as Objects. This is where Wrapper classes come into the picture. A wrapper class is the one which contains or wraps the primitive data types (int, char, etc.)

All the primitive data types have their corresponding Wrapper classes. The following table shows the primitive data types and their corresponding Wrapper class.

| **Primitive data types and their corresponding Wrapper classes** | |
| --- | --- |
| **Primitive Data Type** | **Wrapper Class** |
| **char** | **Character** |
| **byte** | **Byte** |
| **short** | **Short** |
| **long** | **Long** |
| **float** | **Float** |
| **double** | **Double** |
| **boolean** | **Boolean** |
| **int** | **Integer** |

Wrapper classes also provide a number of useful methods to manipulate values. They belong to the java.lang package as part of the Java library.

1. Integer wrappedInt = new Integer(2);

Since Java 5, this can be done without an explicit call to the constructor

1. char value = 'a';
2. Character wrappedChar = value;

This mechanism is called**autoboxing**which means automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of char to Character as shown above.

Similarly, there is a process of **unboxing** which is opposite of autoboxing. In this process automatically conversion of a wrapper class object to its corresponding primitive type is performed. For example refer the code given below.

1. char newVal = wrappedChar;

**Important Note**: Similar to the String pool, Byte, Short, Integer and Long classes cache values in the range -128 to 127.

Next, let us discuss few methods which are present in a wrapper class.

# Wrapper Class Methods

Wrapper classes are helpful in converting numeric strings into numeric datatypes (for example String-to-int, String-to-double etc.). And so, we have supporting methods like parseDouble(), parseInt(), etc.

Each such method accepts a string and returns an object of the corresponding data type. Let's look at the Integer class for instance.

1. String sum = "123";
2. int mySum = Integer.parseInt(sum);*// Here 'mySum' would be holding the integer 123*

Also, typecasting fails in converting a wrapper type to another type. So, we can use methods such as intValue(), byteValue(), floatValue() etc. to do such conversions. Let's take a look at the following example:

1. Integer phoneNo = 44281234;
2. Long phoneNo = phoneNo.longValue(); *// Converts Integer into a Long value*

Now we will see some of the methods of Character wrapper class :

We create two objects alphaObj and digitObj as shown:

1. Character alphaObj = new Character('A');
2. Character digitObj = new Character('5');

Following are some of the methods which make our task to manipulate the data, easy.

1. isDigit() - checks if a given character is Digit
2. isUpperCase() - checks if a given character is LowerCase
3. toString() - converts Character to String
4. charValue() - converts Character to char primitive data type
5. toLowerCase() - converts Character to LowerCase
6. boolean result1 = Character.isDigit(digitObj); *// Output :- true*
8. boolean result2 = Character.isUpperCase(alphaObj); *// Output :- true*
10. String val = alphaObj.toString(); *// val = "A"*
12. char beta = alphaObj.charValue(); *// beta = 'A'*
13. char c = Character.toLowerCase('B'); *// c = 'b';*

We will see some methods which introduce comparison.

1. int result11 = Character.compare('A', 'b');
2. System.out.println(result11); *//-33*
4. Character anotherCharacter = new Character('b');
5. int result22 = alphaObj.compareTo(anotherCharacter);
6. System.out.println(result22); *//-33*

As you can see, both the methods are giving the same output then, what is the difference between them?

If you observe both the methods carefully you will come to know, in compare() we are comparing the two char values (In this case 'A' and 'b') and in compareTo() we are comparing two objects of Character wrapper class.

The working of both the methods is given below.

1. compare() :-

* Comparing chars using compare which returns an int
* It returns 0 if char1 == char2
* a value less than 0 if char1 < char2
* a value greater than 0 if char1 > char1

2. compareTo() :-

* Comparing Character objects using compareTo which returns int
* It returns 0 if obj1 == obj2
* a value less than 0 if obj1 < obj2
* a value greater than 0 if obj1 > obj2

You will explore more of wrapper class methods in the tryout.

Array In Java:

An **array**is a collection of similar data in contiguous memory locations referred to by the same name. It is an object in Java and is created dynamically.

**Syntax:**

1. <data type><variable name>[ ] = new <data type>[size];
2. (or)
3. <data type>[ ] <variable name> = new <data type>[size];

Creating, initializing and accessing an array:

 We initialize array pizzaCost and add elements to it as shown below.

1. int pizzaCost[]=new int[3];
2. pizzaCost[0]=100;
3. pizzaCost[1]=250;
4. pizzaCost[2]=390;

To access each element in the array, we are going to use a for loop as shown.

1. *//Accessing elements using for loop*
2. for (int index; index < pizzaCost.length; index++) {
3. System.out.println(pizzaCost[index]);
4. }

You can use different loops to iterate and access each element of the array. Example of forEach loop is shown below

1. *//Accessing elements using for-each loop*
2. for (int cost : pizzaCost) {
3. System.out.println(cost);
4. }

The type of arrays which we have learned is one dimensional, can we have multi-dimensional arrays?

Have you ever tried to store the data in a matrix?

The solution of the same you will get in the next module.

Multidimensional Array:

We can also have multi-dimensional arrays. In simple words, we define multidimensional arrays as an **array of arrays**. Data in the mutli-dimensional array is stored in the tabular form (you can think of a matrix for better understanding).

**Syntax:**

1. <data\_type>[1st dimension][2nd dimension][]...[nth dimension] <arrayName> = new <data\_type>[size1][size2]….[sizeN];

Now we will see the examples of declaring a multi-dimensional array.

**Examples:**

1. Two dimensional array: int[][] 2dArray = new int[10][20];
2. Three dimensional array: String[][][] 3dArray = new String[10][20][30];

Declaration of the array given above is the direct method of declaration, here we specify the size at the time of declaring an array. To set the values we make use of the dimensions(rows and columns). For example, see the code given below.

1. String[][] names = new String[7][3]; *// Here, 7 is the row size; 3 is the column size (optional)*
2. *// To set value of first row second column*
3. names[1][2] = "Oliver";
4. *// To set value of second row first column*
5. names[2][1] = "Josh";

Now you will see the indirect method of declaring a 2-D array. Consider the array given below having four rows and three columns

1. *// Another way of creating a 2-D array*
2. String[][] names = new String[][] {
3. {"Robert", "Shelly", "Scarlett"},
4. {"Mark", "Oliver", "Chris"},
5. {"Emmy", "Josh", "Merry"},
6. {"Evelyn", "Austin", "Emma"}
7. };

To access any element of the array we refer its position i.e. arrayName[row][column].For example

1. String value = names[0][1];
2. *// value = Shelly*

 Now you will learn how to iterate over the multi-dimensional array.

1. for(int i=0;i<names.length;i++) {
2. for(int j=0;j<names[i].length;j++) {
3. System.out.print(names[i][j]+" "); *// where i refers to row*
4. *// and j refers to column*
5. }
6. System.out.println();
7. }

As you can see, we have used two for loops to iterate over array(names). Outer for loop is used to iterate each row of the 2D array (for better understanding consider it as a 2D matrix) and the inner one iterates over each element in the row.

Now as we have learned the concept of array thoroughly, can we create an array of user-defined objects?

What kind of array that will be? Let us see that next.

ARRAY OF OBJECT:

An object represents a single record in memory, consider a scenario where we want multiple records (objects) at a time, in that case, you will have to use arrays which will be called as an array of objects. Make a note, arrays can hold the only reference of the objects and not the object. Let us see how to create an array of objects.

**Example:**Consider a class Account consisting of two attributes account holder's name and its bank account number. We have to show the information of each account holder present in the bank. To make operation easy we have to make use of arrays.

First, in order to use an object, you should define the structure of the class Account.

1. class Account{
2. String name;
3. long accountNumber;
4. }

Now to perform the required operation we have to define two methods, one for setting the data and others for displaying the information. These methods will be called by the objects every time, to perform the operation. Accordingly, the structure of the class Account will become as given below:

1. class Account{
2. String name;
3. long accountNumber;
5. void setData(String custName,int accNum) {
6. this.name = custName;
7. this.accountNumber = accNum;
8. }
9. void display() {
10. System.out.println("Name :- "+name);
11. System.out.println("Account Number :- "+accountNumber);
12. }
13. }

 Next task is to declare an array, which is shown below:

1. *// creating array of type Account having size 3*
2. Account accArray[] = new Account[3];

After completing the declaration, now comes the part where we have to create an object and set the data (account holder's name and account number).

1. *//first create the object of Account*
2. *//and then set the data for each object*
3. accArray[0] = new Account();
4. accArray[0].setData("Ella",345234);
6. accArray[1] = new Account();
7. accArray[1].setData("Scarlet",345278);
9. accArray[2] = new Account();
10. accArray[2].setData("Harper",345897);

We can call the method display using these objects individually (accArray[0].display()) as the total number of objects is 5, but what if we have more than one thousand objects? Therefore we make use of loops, enhanced for loop is preferred as it is easier than traditional for loop. See the code given below to get the grip of an idea.

1. System.out.println("Account holder's information");
2. System.out.println();
3. for (Account account : accArray) {
4. account.display();
5. System.out.println();
6. }

You will further explore this concept in tryouts.

ARRAYS class:

Java also has a inbuilt class named Arrays which contains various methods for performing operations like searching and sorting on arrays.

**Arrays**is an inbuilt Java class belonging to **java.util** package. It has many static methods for quick comparing, searching, sorting of arrays and other utility methods. Many of these methods help us to do away with loops required for manipulating and accessing arrays.

Arrays is an inbuilt class and should not be confused with an array which is collection of homogeneous elements.

Methods of the class can be called using below syntax –

Arrays.<<method name>>

Example

1. int[] numArr = {6,8,9,10,40,66};
2. int resultIndex = Arrays.binarySearch(numArr, 40);

Here index of the search element will be returned. A negative index is returned if element is not found.

Another useful method is **equals()** -

1. int[] numArr1 = {6,8,9,10,40,66};
2. int[] numArr2 = {6,8,9,10,40,66};
3. boolean resultCompare = Arrays.equals(numArr1, numArr2);
4. System.out.println(resultCompare); *//prints true or false*

To sort an array we just need to do the following-

1. int[] numArr = {6,8,9,10,40,66};
2. Arrays.sort(numArr);

**sort()** method does not return any value and changes the content of original array.

Arrays class provides its own version of **toString()** method which can be used to quickly print elements of an array.

1. int[] numArr = {6,8,9,10,40,66};
2. System.out.println(Arrays.toString(numArr));

You can explore more about Arrays [in the Oracle docs on Class Arrays](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html).

Enumeration:

Enumerations are group of named constants. All enums implicitly extend the java.lang.Enum class. The enum fields are implicitly static and final, and hence are constant during compile time. But they are instances of their enum type, constructed when the enum type is referenced for the first time.

**Why do we need these grouped constants?**

Let us assume that a developer is creating online Pizza ordering application. He wants to allow the customers to choose the size of the pizza. The sizes allowed are small, medium and large only.  
He realizes that having the type of the size variable as String has a chance of some developer entering any arbitrary size. And this could cause invalid processing. How can we stop the Pizza size to be initialized anything other than these three values.

Using enums allows us to limit the selection within a set of values.

An **enum** is a datatype which contains a fixed set of constant values. For example,

* directions (NORTH, EAST, WEST, SOUTH)
* pizzaSize (SMALL, MEDIUM, LARGE)

Syntax:

1. public enum enum\_name { constant1, constant2, ..., constant n }

Example:

1. public enum PizzaSize { SMALL, MEDIUM, LARGE }

This type can be declared  to limit the usage to the above 3 values.

1. private PizzaSize size;

Let us see what all things can be done with enum based variables next.

Let us look at some features about enums:

* Enums are considered as reference-types like classes and interfaces in Java, and hence, a programmer can define constructors, methods and variables, inside them.
* A static method called **values()**is automatically generated by the Java compiler for each enum. The **values()** method returns an array of all the constant values defined inside the enum.
* Enum variables can be used in an if statement or switch statement.

**if statement-**

1. if(this.size.equals(PizzaSize.MEDIUM)){
2. System.out.println("Size is Medium");
3. }

**switch statement-**

1. PizzaSize currentSize = PizzaSize.MEDIUM;
2. double discount = 0;
3. *// using enum in switch case*
4. switch (currentSize) {
5. case SMALL:
6. discount = 10;
7. break;
8. case MEDIUM:
9. discount = 20.5;
10. break;
11. case LARGE:
12. discount = 30.2;
13. break;
14. }

We can loop through the enum and print constant values.

**for loop-**

1. *//values() method returns an array of all values inside enum*
2. *//ordinal() method can be used to display values assigned to enum constants*
3. for (PizzaSize psize : PizzaSize.values()) {
4. System.out.println(psize+" "+psize.ordinal());
5. }

Code Analysis:( Sonar Lint)

It is a well-known fact that standardization creates a positive impact on business. In the Software Industry, it is important to follow coding standards for any software applications to run the smooth and successful functioning of the organizations.

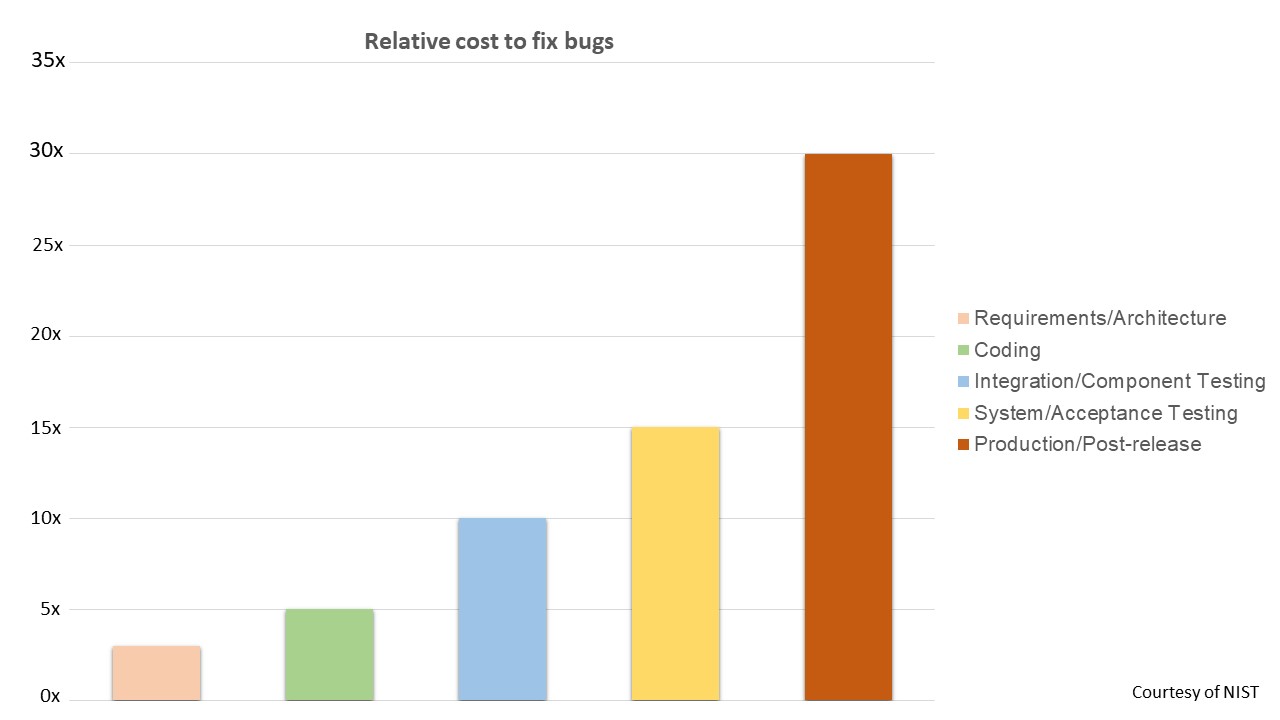
A good programmer is one who writes good code, which can be independent enough to re-use in multiple ways. Good code reduces defects, reduces security vulnerabilities in the application, it is well documented which should be easy to review and test. For a piece of code to be considered as good, it should follow coding standards.

What are the coding standards?

**Coding standards** are a series of specific guidelines that need to be followed while coding like programming style, naming conventions for classes & methods, and various other aspects of the applications. Coding standards play a major role in software development. It provides easily understandable, maintainable, and consistent code.

Lack of coding standards affects the quality of the application.

A **bug**is a piece of code that can and will break your code. The below image shows how much it costs to resolve bugs at various stages of a sample application. In the below image the x-axis represents the number of times the cost has been multiplied.



Benefits of following coding standards:

* Efficient code
* Less risk for project failure
* Consistency will give code clarity
* Since the code is consistent, bugs can be traced easily
* Cost-effective, finding and resolving bugs after release is a costly process

Next, we will see what code quality is and the different tools to measure code quality.

As an application developer, we will be writing hundreds of lines of code in hundreds of files. Quality of your code defines how safe, secure, and efficient your application is. Hence the quality of the code written should be given as much importance as that of the most important feature of your application.

**Code quality**is a measure of how good or bad your code is. It is not a fixed value/standard, it will differ for different teams as different applications have different requirements. So a team has to define its custom standard or requirement for how good their code should be. They should also ensure that all the team members adhere to these standards strictly so that the final product is of the highest quality.

There are multiple traits one should find in the code to achieve the highest quality. Here are a few key traits that every good code should have.

* **Reliable:**The code should be highly reliable, this is the measure of how long an application runs without failure.
* **Maintainable​​​​​​:**It refers to how easy it is to maintain an application.
* **Testability:** It refers to how easy it is to test individual components of any application.

There are multiple tools available in the market to measure code quality like:

* PMD
* Checkstyle
* FindBugs
* SonarQube

**SonarQube**is a server where we can host our projects and run analysis, it internally uses all the above-mentioned tools to detect issues and generate reports on the code quality. This will be seen in future courses.

**SonarLint**is a plugin/extension that allows developers to connect to the SonarQube server and run the analysis.

In this course, we will learn how to use SonarLint in Eclipse IDE.

**SonarLint** is an extension/plugin that can be installed on an IDE ( Integrated Development Environment ) like Eclipse or Visual Studio Code to check for code quality on-the-fly. It scans the code being written for violations and suggests solutions to the developer dynamically like a spell checker. Since the code being written is checked on-the-fly, the productivity of the developer is increased many-fold.

Since it is installed as a plugin/extension, all the rules will be available offline, the developer need not be connected to a network while he/she is writing code. If a team of developers has a very specific requirement for its application, the developers can create a custom ruleset on the **SonarQube**server to which all other team members can connect with using the **SonarLint** plugin and follow the custom rules instead of the default rules. This ensures that the code is uniform throughout the application which will be beneficial in maintaining the application.

Here are some benefits of using SonarLint,

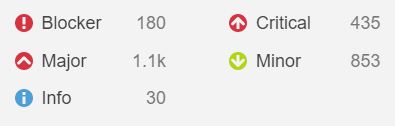
* Rules are stored offline, hence internet connectivity is not required.
* On-the-fly analysis ensures better code and increased productivity.
* It can be customized to ignore unwanted files.
* Rules can be customized to match the requirements of the projects specifically.

SonarLint has categorized the different violations a developer will/can do into different types, here are a few.



* **Bug:**It is a part of the code that will break the execution of the application abruptly.
* **Vulnerability:** It is a part of the code prone to external attacks leading to loss of critical information.
* **Code Smell:**It is a part of the code written in an unnecessarily complicated manner leading to maintenance issues in the future.
* **Security Hotspot:** Similar to a vulnerability, this may or may not lead to loss of user information, It is up to the developer to either fix this immediately or later on.

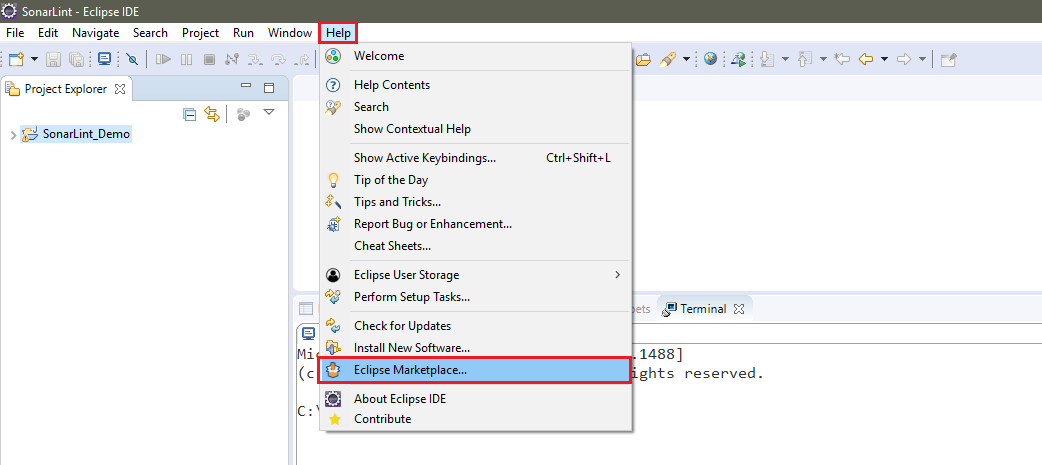
Violations are also categorized based on their severity to the application execution(See the [SonarCube docs on Issues](https://docs.sonarqube.org/latest/user-guide/issues/" \t "_blank) to know more about the violations),



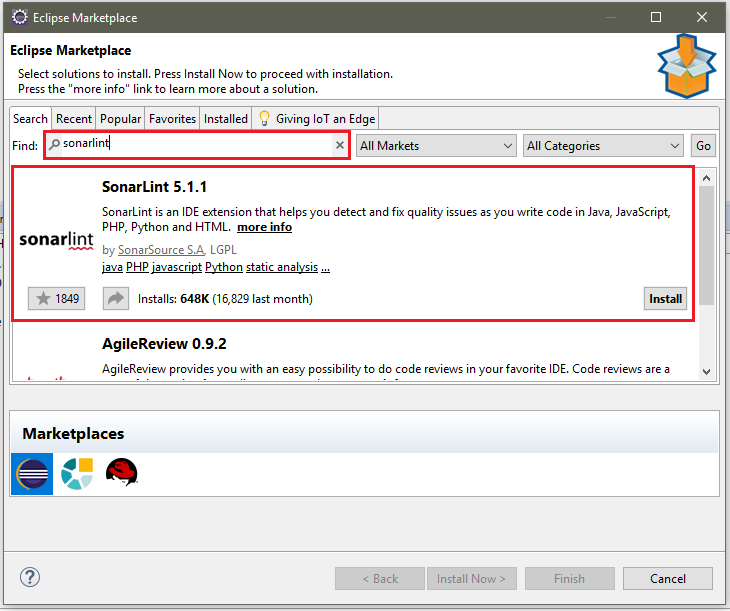
The developers can change the severity of a violation in their custom rules. Next, we will learn how to install and use SonarLint plugin in eclipse.

Steps to install **SonarLint**in Eclipse IDE.

**Step 1:**Launch the IDE and click on**Help > Eclipse Marketplace**



**Step 2:**In the pop-up window, Search for **SonarLint**and click on **Install** on the search result.

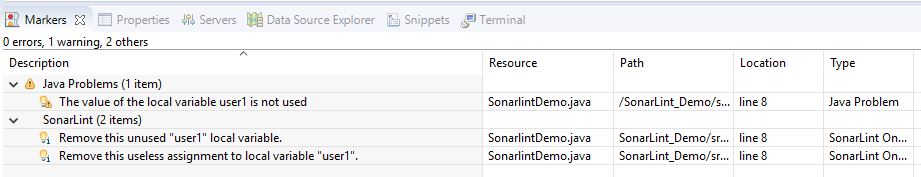


Note: The version of SonarLint is 5.1.1 during the installation. When you try to install it in your eclipse, the version might change.

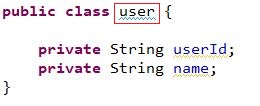
Once the extension is installed, agree to the **terms and conditions** on the next page and click on finish. Restart eclipse(if not prompted) to enable the extension.

We are going to check for code quality using **SonarLint**

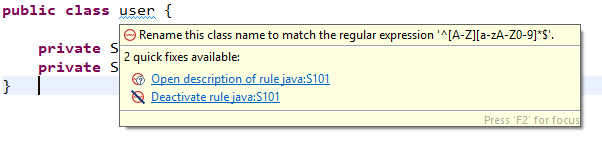
Once eclipse restarts, **SonarLint** will start scanning (by default) the files open in your workspace and list out all the violations/issues in the **markers** view as shown below.



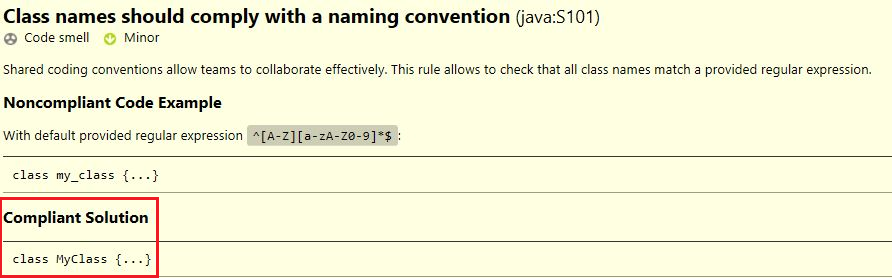
Since, **SonarLint**checks for issues on-the-fly, it will dynamically provide feedback on what you're coding. SonarLint will highlight the issue and provide information on what is wrong. For example, the **user**class as defined below is violating the class naming convention.



On hovering over the violation being highlighted by SonarLint, we get the below screen,

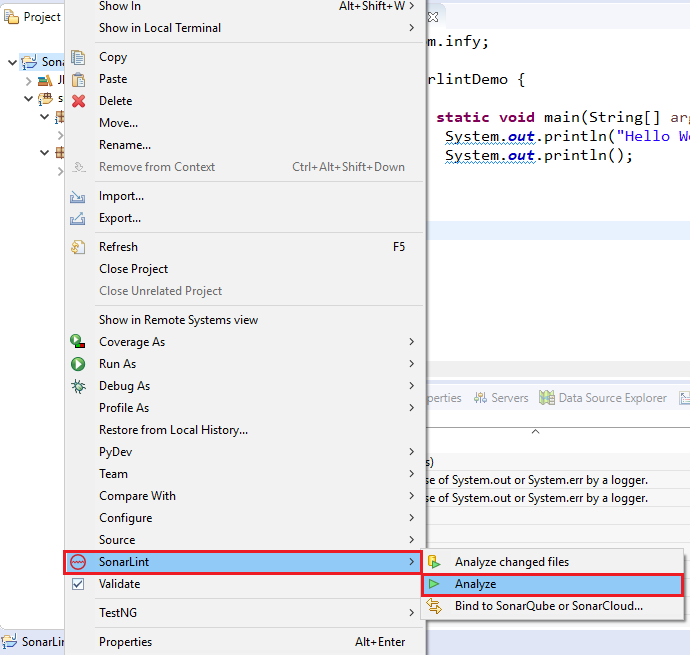


Click on the **Open description** link to see more detailed information on the rule. This description will include the reason as to why the usage is considered incorrect as well as a better and more compliant alternative that can be considered.

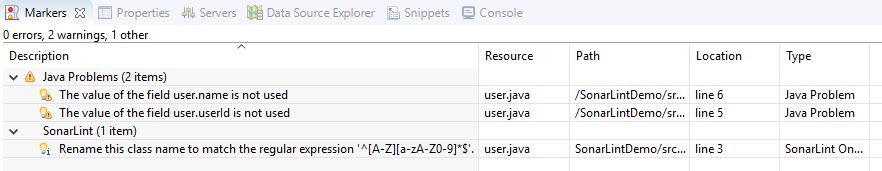


Click on **Deactivate Rule,**if a rule should be ignored for the rest of the code.

To manually check for code quality, **Right Click** on the project and click on **SonarLint**> **Analyze**



A new **SonarLint Report** view will be displayed as shown below. Click on the description to view detailed information on the rule.



You have seen one of the many rules of SonarLint for Java. See the [Sonarsource page on rules for JAVA](https://rules.sonarsource.com/java" \t "_blank) to view the complete ruleset.

In our course, there are a few rules of Sonarlint that we may not follow. The most prominent one of them is the use of System.out statements. Let us see what it is and why we should avoid its usage.

Till now, we have displayed the data of our Java projects in the console of the Eclipse IDE using the System.out.\* statements. But according to coding practices, this is a major violation.

What are these statements and why are they considered as major violations?

**System.out.\*** statements are usually used when the developer wants to log some information related to the application on the console. But it is considered bad practice since the information logged by the application and the server the application is running on will log on the same console and it will become highly impossible to separate between the two.

Here are a few other reasons why a good developer should not use **System.out** statements in their code.

* If the application/server stops, the information logged on the console is lost.
* The developer should be able to access the logged information whenever required, hence the use of a separate log file is recommended.
* The level of information being logged cannot be set when using System.out. So there is no difference between genuine information and an error message being logged.
* There is no fixed format for the information being logged, so it is not easy for newer developers to understand what is being logged.

In this course, we will be using console-based applications for demos and assessments that require the developers to log certain data to the console, unlike an enterprise application that runs on a server. The data being logged is an indication for success or error status of the application. Hence System.out statements will be used in a limited capacity.

If a developer has a requirement to log information from an enterprise application, it is recommended to use frameworks specifically designed to overcome the issues of the System.out statements like **Log4j2**, **Apache Commons Logging**, **SLF4J**, etc.

# Assignment-1

# Problem Statement:

Quadratic equation is an equation with degree 2 in the form ax2 +bx + c = 0, where a, b, c are the coefficients. Write a program to solve a quadratic equation.

Your output will be decided based on the value of the discriminant (which is b2 -4\*a\*c),

* If the discriminant value is 0, then both the roots will be equal. Print the value of the root obtained.
* If the discriminant is greater than 0, then both of the roots are real and distinct. Print both the values.
* If the discriminant is less than 0, then the equation will have no real roots. Print a message which conveys that the roots are not not real.

### ****Sample I/O table:****

Please refer the given table for sample input values and its corresponding output.

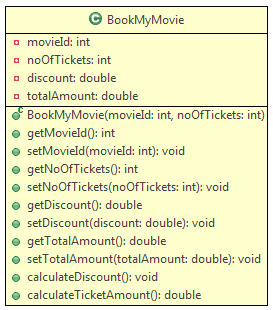
| ****Input/Output**** | |
| --- | --- |
| ****Input**** | ****Output**** |
| **a=1, b=4, c=4** | **The root is -2.0** |
| **a=2, b=5, c=2** | **The roots are -2.0 and -0.5** |
| **a=1, b=4, c=6** | **The equations does not have real roots.** |

**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

# Assignment-2:

# Problem Statement:

BookMyMovie is an online website, through which the required number of tickets can be booked for a particular show. A maximum number of 15 tickets can be issued per booking. Implement the class BookMyMovie as per the class diagram given below:



### ****Method Description:****

* **BookMyMovie(int movieId, int noOfTickets):** This constructor is used to initialize the instance variables with user provided values.
* **calculateDiscount():** This method is used to set the discount percentage for a particular booking based on the ***noOfTickets***issued and the ***movieId***. The condition for calculating the discount is given below:

| ****Movie Data**** | | |
| --- | --- | --- |
| ****MovieId**** | ****noOfTickets**** | ****Value**** |
| **101, 102, 103** | **<5** | **0** |
| **101 or 103** | **>=5 && <10** | **15** |
| **101 or 103** | **>=10 && <=15** | **20** |
| **102** | **>=5 && <10** | **10** |
| **102** | **>=10 && <=15** | **15** |

​​​​​​​

Note: In this above table, Value column stands for the discount percentage.

* **calculateTicketAmount():** This method is used to calculate the ticket amount after deducting the discount amount. It should call the method ***calculateDiscount()*** which sets the discount percentage. Then the ***totalAmount***should be calculated using the formula given below:
  + totalAmount = baseFare \* noOfTickets – (baseFare \* noOftickets \* (discount/100))

The baseFare for a given movieId is given below:

| ****Movie Data**** | |
| --- | --- |
| ****MovieId**** | ****Base Fare**** |
| **101** | **120** |
| **102** | **170** |
| **103** | **150** |

Once you implement the above class,

* Create a Tester class.
* Create objects of BookMyMovie class with the sample inputs given below.
* Invoke the **calculateTicketAmount()** for each of the object and verify with the sample output provided.

### ****Sample I/O:****

* **Input**(For BookMyMovie object 1):
  + movieId = 101
  + noOfTickets = 5
* **Output**(For BookMyMovie object 1): Total amount for booking: 510.0

* **Input**(For BookMyMovie object 2):
  + movieId = 102
  + noOfTickets = 4
* **Output**(For BookMyMovie object 2): Total amount for booking: 680.0

* **Input**(For BookMyMovie object 3):
  + movieId = 103
  + noOfTickets = 8
* **Output**(For BookMyMovie object3): Total amount for booking: 1020.0

* For any invalid movieId, the output should be,

1. Sorry! Invalid Movie ID!
2. Please check the Movie ID and enter once again.

**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

**ASSIGNMENT-3:**

**XXXAcademy**is an online academy which invites candidates to participate in various events, and also provides grades for each candidate based on his/her performance. They need your help in two modules,

* Adding candidates
* Getting candidate reports

| **Artifacts:** | |
| --- | --- |
| **File Name** | **Status** |
| **com.infy.dao.CandidateDAO.java** | **Implemented** |
| **com.infy.model.Candidate.java** |
| **com.infy.userinterface.CandidateTester.java** |
| **com.infy.model.CandidateReport.java** | **To be Implemented/modified** |
| **com.infy.service.CandidateService.java** |
| **com.infy.test.CandidateServiceTest.java** | **To be ignored for now** |
| **com.infy.validator.Validator.java** |

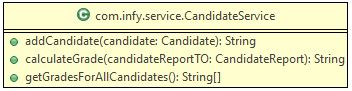
**CandidateReport:**

Please implement the class based on the class diagram given below. Please follow the class diagram strictly.



Make sure that you keep the variable names as it is shown in the class diagram.

**CandidateService:**



This class has three methods which you have to implement based on the description given below,

**addCandidate(Candidate candidate):**

This method accepts a Candidate object, validates the marks and result of the candidate and sends it to the CandidateDAO class to be inserted to the database table.

* If any one of the marks attribute of the Candidate object is below 50, and the result attribute of the Candidate object is '**P'**, return the String message- "**Result should be 'F' (Fail) if student scores less than 50 in any one subject".**
* Else, invoke the addCandidate() from CandidateDAO class by passing the Candidate object as the parameter, which will return a String value.
* Return the String value obtained in the previous step.

**calculateGrade(CandidateReport candidateReportTO):**

This method calculates the average of the marks in the CandidateReport object, and return a grade relevant to the average.

* If the result of the CandidateReport object is **F**, return **NA**
* Else, calculate the average of the marks attribute of the CandidateReport object.
* Return the grade based on the average obtained in the previous according to the table given below,

| **Data** | |
| --- | --- |
| **Average** | **Grade** |
| **>=85** | **A** |
| **<85 and >=75** | **B** |
| **<75** | **C** |

**getGradesForAllCandidates():**

This method gets all the CandidateReport details from the CandidateDAO, creates a String value based on the CandidateReport object's candidateID and the grade obtained by invoking the calculateGrade() method of the CandidateService class.

* Invoke the getAllCandidates() method from the CandidateDAO class which will return a CandidateReport array.
* Iterate over the array obtained in the previous step and create a String value for each of the object in the below format,

**candidateId:grade**

where, **grade** is obtained by invoking the calculateGrade() method of the CandidateService class by passing the CandidateReport object as the parameter.

* Add all the created String values in a new String array.
* Return array created in the previous step.

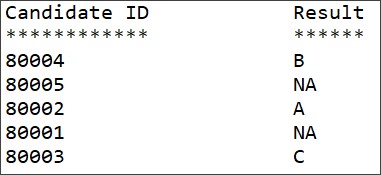
**Sample Input and Output:**

**Add Candidate**

| **Input Data** | |
| --- | --- |
| **Attribute** | **Value** |
| **candidateId** | **12346** |
| **candidateName** | **Sam** |
| **mark1** | **51** |
| **mark2** | **56** |
| **mark3** | **78** |
| **result** | **P** |
| **department** | **ECE** |
| **examDate** | **29th May, 2014** |

* **Output:** Candidate details are successfully added.

**Get Candidates Results:**

* **Output:** ​​​​​​​​​​​​​​​​​​​​​​​​ ​​​​​​​​​​​​​​ ​​​​​​​

**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

Static Keyword:

Ford, a car manufacturing company, wants to keep a count of the cars manufactured. The class below creates a car based on the model and color and also keeps their count.

1. class Ford {
2. private String modelNo;
3. private String color;
4. private int noOfCars; *// For counting cars*
5. public Ford(String modelNo, String color) {
6. this.modelNo = modelNo;
7. this.color = color;
8. this.noOfCars++; *// Incrementing the count*
9. }
10. }

Each time the company receives an order for a new car, we expect the value of noOfCars to be incremented.  
But it is not working!! Can you figure out why?

… because for every object of the class, a new copy of all the instance variables and methods are created.

This calls the need for a kind of variable that can be shared and manipulated by all the instances of a class. Java's solution to this is the **static**keyword.

**Static keyword** is used to make a member belong to a class, and not to any of its individual objects. Only one copy of the member is maintained across all the instances. We can have static variables, methods, and blocks.

The static variables are *class level variables* that are used to keep a value across all the instances of a class. They are initialized when the class gets loaded.

The class Ford can now be modified to meet the requirements:

1. class Ford {
2. private String modelNo;
3. private String color;
4. private static int noOfCars; *// Creates a class variable*
5. public Ford(String modelNo, String color) {
6. this.modelNo = modelNo;
7. this.color = color;
8. noOfCars++; *// For every object created, the value of the same variable gets incremented*
9. }
10. }

Now let's see how static methods and static blocks can be useful…

**Static methods** are *class level methods*. They can be used without any instance of the class, and are invoked using the class name. They can be called using a reference of the class as well.

Let's say, instead of creating an object of Ford ourselves, we want to place an order for it and have Ford make it for us. We just want to specify the model and color.

Ford's method to do this has to be static as it needs to be common for all.

1. public static Ford orderCar(String model, String color) {
2. Ford car = new Ford(model, color);
3. return car;
4. }

The method call will look something like this:

1. public static void main(String args[]) {
2. Ford myMustang = Ford.orderCar("Mustang", "Red");
3. System.out.println("Thanks for ordering a Ford ...Drive safe!");
4. }

**What is a Static Block?**

  Java supports a special block, called **static block** which gets executed when the class is loaded in the JVM. A static block is used for initializing static variables.

The code in the static block will get executed, once either of the below condition satisfies, when

* The object is created
* The first time a static member of that class is accessed.

A Class can have multiple static blocks, which will execute in the same sequence in which they have been written.

**Why do we need Static Block?**

       If a class has static members which require complex initialization, in that case, the static block is the solution.

**Rules of Static Block**

* Static block cannot return a value.
* Static blocks get executed before constructors.
* Static block is used to initialize static variables only; it gives compilation error if we try to initialize non-static variables.
* Keywords like this and super cannot be used inside the static block.

Let's take an example. Suppose we need to automatically generate the driver IDs in the Driver class, starting from some initial value:

1. private static int counter;
2. static {
3. int randomNumber = 30;
4. counter = (int) Math.pow(randomNumber, 3);
5. }

This static counter variable can now be used to initialize the driver IDs of the drivers:

1. public Driver() {
2. this.driverId = counter++;
3. }

Static block cannot be called explicitly.

Now have a look at this,

1. class Ford {
2. private String modelNo;
3. private String color;
4. private String bookingId;
5. private static int noOfCars;
6. public static void generateBookingId() {
7. this.bookingId = "F" + noOfCars; *// Compilation Error! Why?*
8. }
9. }

The static method doesn't belong to any object, so it does not know a member of which object we are trying to access. Trying to do so will result in a compilation error.

A **static context**, i.e. static blocks and static methods, cannot access non-static (instance) members directly.

However, non-static methods can access static members.

Method Overloading:

When we need different implementations of the same behavior depending upon the context, we go for method overloading.

Method overloading lets you have more than one method with the same name in a class. The methods accept parameters differing in their data types, the number of parameters, or their order.

Let's say we want to search for cars in a car store based on certain criteria, like manufacturer, color, etc. We can have overloaded methods to search for cars.

1. class CarStore {
2. public void searchCar(String manufacturer) {
3. *// finds cars from the desired manufacturer*
4. }
5. public void searchCar(float topSpeed) {
6. *// finds cars having the desired top speed*
7. }
8. public void searchCar(float topSpeed, String color) {
9. *// finds cars having the desired top speed and color*
10. }
11. }
12. class Tester {
13. public static void main(String[] args) {
14. CarStore showroom = new CarStore();
16. showroom.searchCar("Nissan");
17. showroom.searchCar(180.0, "Blue");
19. *// Rest of the code*
20. }
21. }

Which searchCar method will be invoked?

Call to overloaded methods is resolved during **compile time** based on the method signature. Hence, called **static polymorphism**.

Constructors can also be overloaded.

Just like normal methods, constructors can also be overloaded, i.e., a class can have multiple constructors. This is called constructor overloading. Remember that you have already seen a class having multiple constructors.

The below code demonstrates constructor overloading.

1. class Customer {
2. private String customerId;
3. private String customerName;
4. private long contactNumber;
6. *// Parameterless constructor*
7. public Customer() {
8. System.out.println("Inside parameterless constructor");
9. }
10. *// Parameterized constructor*
11. public Customer(String customerId, String customerName, long contactNumber) {
12. this.customerId = customerId;
13. this.customerName = customerName;
14. this.contactNumber = contactNumber;
16. }
17. *// Parameterized constructor*
18. public Customer(String customerName, long contactNumber) {
19. this.customerName = customerName;
20. this.contactNumber = contactNumber;
22. }
23. *// Methods including getter and setter and other methods*
24. }
25. public class Tester {
26. public static void main(String[] args) throws Exception {
27. */\**
28. *Parameterless constructor will be invoked, instance variables should*
29. *be initialized using setter methods*
30. *\*/*
31. Customer customerOne = new Customer();
32. */\**
33. *Parameterized constructor with three parameters instance variables are*
34. *initialized in the constructor*
35. *\*/*
36. Customer customerTwo = new Customer("C1016", "Stephen Abram",
37. 7856341287L);
38. */\**
39. *Parameterized constructor with two parameters instance variables*
40. *are initialized in the constructor*
41. *\*/*
42. Customer customerThree = new Customer("James Jonathan", 7828171287L);
43. }
44. }

Constructor Overloading is the combination of two concepts, Constructor and Method Overloading. It means defining multiple constructors for the same class. We know, the constructor is used for initializing the object but in some cases, we need to create multiple objects with the various numbers of parameters.

Important points related to Constructor Overloading:

* Constructor call must be the first statement in the constructor
* To invoke the default constructor of the class we use this()

Implementation of these rules is given below:

1. public Customer(String customerName, long contactNumber) {
2. this.customerName = customerName;
3. this.contactNumber = contactNumber;
4. }
5. public Customer(String customerId, String customerName, long contactNumber) {
6. this(customerName,contactNumber); *// Will give error if not first statement in constructor.*
7. this.customerId = customerId;
9. }

To initialize the name and contact number again we make use of this keyword where this(customerName,contactNumber) will invoke the constructor with these (customerName,contactNumber) parameters and initialize the attributes.

super() is used to invoke parent class constructor, refer the tryout given in Inheritance module.

Inheritance:

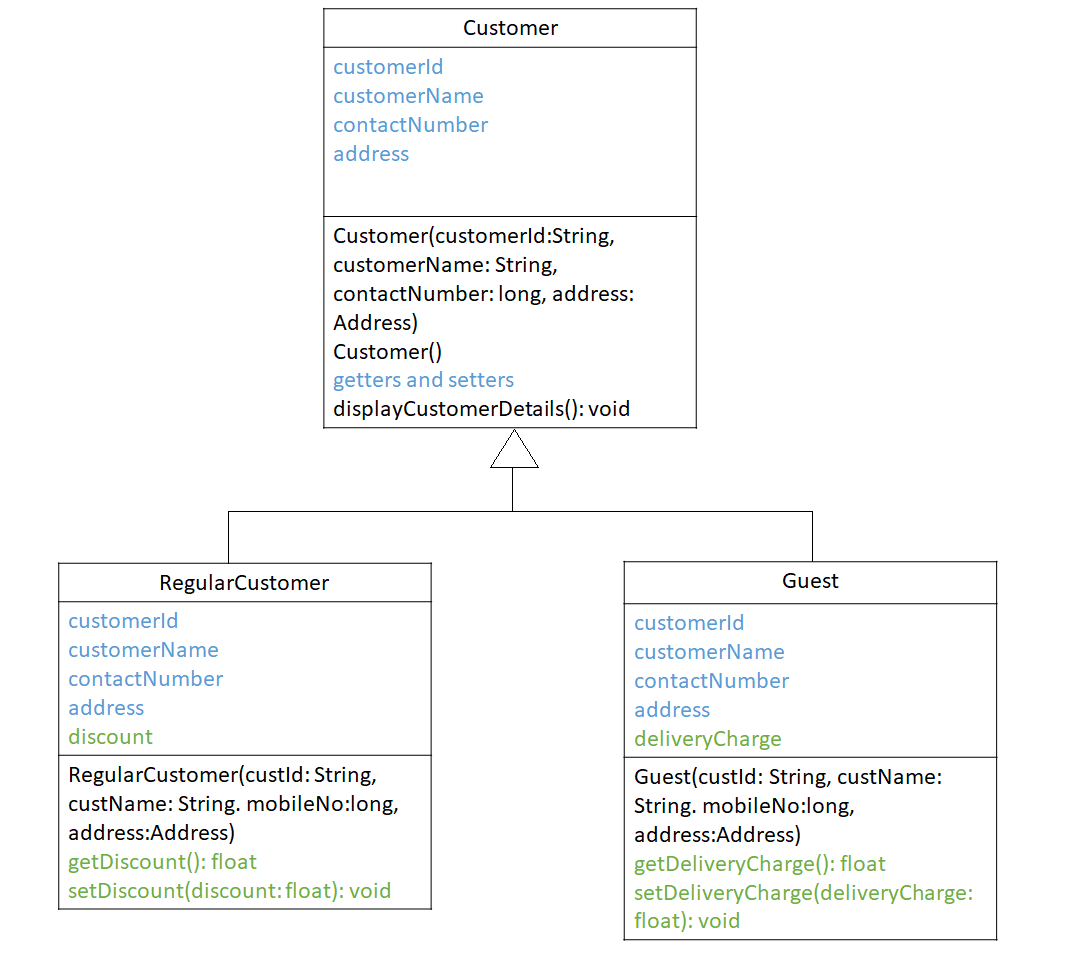
You will now have a look at one more relationship type.

Let us consider the scenario of SwiftFood app. The customers of SwiftFood can be divided into three types: Regular, Premium and Guest. Regular and Premium customers are eligible for 5% and 8% discounts respectively on their orders. Also, premium customers are provided with premium cards so that they can redeem points while ordering food. Guests pay delivery charges for their orders.

Though Regular customers, Premium customers and Guests have different billing mechanism, they have some common attributes like customerId, customerName, contactNumber and address.

So, instead of keeping the common attributes in each and every class, we can have a common class called Customer and include the common attributes.

Now, let us consider RegularCustomer and Guest classes. Since RegularCustomer and Guest are types of Customers, they must have access to the common attributes also. We can create a Customer class with the common attributes and methods and make these two classes inherit the attributes and behavior from Customer class as shown below.



The common members that are inherited are represented using blue color. The members specific to RegularCustomer and Guest are represented using green color.

We can say that the RegularCustomer "is-A" Customer and Guest "is-A" Customer. **When a class inherits from another class, then those classes are said to have inheritance relationship**. The class which is **inheriting is called the child/sub/derived class** and the **class which is getting inherited is called the parent/super/base class**. Inheritance is also called as "is-A" relationship. Inheritance (is-a) is denoted by a line with an arrow head.

In our example, RegularCustomer and Guest classes are inheriting the Customer class. So, Customer is the parent class and RegularCustomer and Guest are child classes.

# What gets inherited ?

In case of an inheritance relationship, the attributes and behaviors get inherited just like a child inherits certain attributes and behaviors from his/her parent.

In terms of OOP, **a child class inherits all the non-private attributes and methods**.

When we say a child class inherits the attributes and methods, we can treat the attributes and methods as if they are owned by the child class itself.

In Java, a child class inherits the parent class using the "extends" keyword. Observe the below code.

1. class Customer {
2. *//Parent/Super/Base class*
3. }
4. class RegularCustomer extends Customer { *// RegularCustomer is a Customer*
5. *//Child/Sub/Derived class*
6. }
7. class Guest extends Customer { *// Guest is a Customer*
8. *//Child/Sub/Derived class*
9. }

Here, RegularCustomer and Guest are the child classes of Customer hence they can inherit properties of the parent class i.e. Customer.

As you have learned in last topic, child class inherits all the non-private attributes and methods, RegularCustomer and Guest will inherit non-private attributes and methods of the Customer Class.

By now, you have learnt that the derived classes inherit from the base class. You will now see how the child class object is created. As you all know, constructors are invoked while creating objects. When a child class object is created, the child class constructor invokes the parent class constructor before executing any statement present in the child constructor.

Observe the below code and understand the sequence of constructor invocation.

1. class Customer {
2. public Customer() {
3. *// 3: Parent constructor will be executed*
4. System.out.println("Creating a customer...");
5. *// 4: The flow will go back to the child constructor*
6. }
7. }
8. class RegularCustomer extends Customer {
9. public RegularCustomer() {
10. *// 2: This constructor will then call the parent constructor*
11. System.out.println("It is a regular customer!");
12. *// 5: The flow will finally come here*
13. }
14. }
15. public class Tester {
16. public static void main(String[] args) {
17. RegularCustomer regularCustomer = new RegularCustomer();
18. *// 1: This line will be executed first and the flow will go to [2]*
19. }
20. }

Need For super keyword:

Consider a scenario, we have a Customer class consisting of private member variables customerId, customerName, setter and getter methods to set and access the private member variables and displayCustomerDetails() method to display the customer details.

1. class Customer {
2. private String customerId;
3. private String customerName;
4. public Customer() {
5. System.out.println("Parent Default Constructor");
6. }
7. public Customer(String customerId, String customerName) {
8. this.customerId = customerId;
9. this.customerName = customerName;
10. }
12. public String getCustomerId() {
13. return customerId;
14. }
15. public void setCustomerId(String customerId) {
16. this.customerId = customerId;
17. }
18. public String getCustomerName() {
19. return customerName;
20. }
21. public void setCustomerName(String customerName) {
22. this.customerName = customerName;
23. }
24. public void displayCustomerDetails() {
25. System.out.println("Displaying customer details \n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");
26. System.out.println("Customer Id : " + customerId);
27. System.out.println("Customer Name : " + customerName);
28. System.out.println();
29. }
30. }

The RegularCustomer class extends Customer class. Only the Regular customers are provided with a discount of 5% on the total cost, so we have the discount attribute in the RegularCustomer class. The common attributes like customerId, customerName, contactNumber and address are not included here. Those attributes will be inherited from the parent class. Observe how the values are being set for all the variables.

1. class RegularCustomer extends Customer {
2. private float discount;
3. public RegularCustomer(String custId, String custName) {
4. this.setCustomerId(custId);
5. this.setCustomerName(custName);
6. this.discount = 5.0f;
7. System.out.println("Child Constructor");
8. }
9. public float getDiscount() {
10. return discount;
11. }
12. public void setDiscount(float discount) {
13. this.discount = discount;
14. }
15. }

For executing above code we will write a Tester class as shown,

1. public class Tester {
2. public static void main(String[] args) {
3. RegularCustomer regularCustomer = new RegularCustomer("C1010",
4. "Johns Kora");
5. regularCustomer.displayCustomerDetails();
6. }
7. }

In the Tester class, we are creating the child (RegularCustomer) class object. First, the control will go to RegularCustomer constructor. The flow will then move to parent (Customer) class constructor. After execution of the parent class constructor, the child class constructor will get executed.

Output:

1. Parent Default Constructor
2. Child Constructor
3. Displaying customer details
4. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
5. Customer Id : C1010
6. Customer Name : Johns Kora

In the previous code snippet, the member variables were being set in the child class constructor and the parameterized constructor of parent class.

Instead of setting the values of member variables in both the places, it would be better if the parameterized constructor of parent class is invoked from the child class constructor so that code can be reused.

By now, you know that the parent class parameterless constructor is implicitly invoked by the child class constructors but how are parameterized constructors of parent class invoked?

This can be done using super as shown in the code given below.

1. public RegularCustomer(String custId, String custName) {
2. super(custId,custName); *//Invoking the parent class parameterized constructor*
3. this.discount = 5.0f;
4. System.out.println("Child Constructor");
5. }

Here, super(custId,custName) will send the parameters to the parameterized constructor written in the parent class which will set the attributes for the current object, and we will get the output as follows:

1. Child Constructor
2. Displaying customer details
3. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
4. Customer Id : C1010
5. Customer Name : Johns Kora

You already know that there are 4 access modifiers in Java. One of them is **protected**.

Let us revisit protected access modifier.

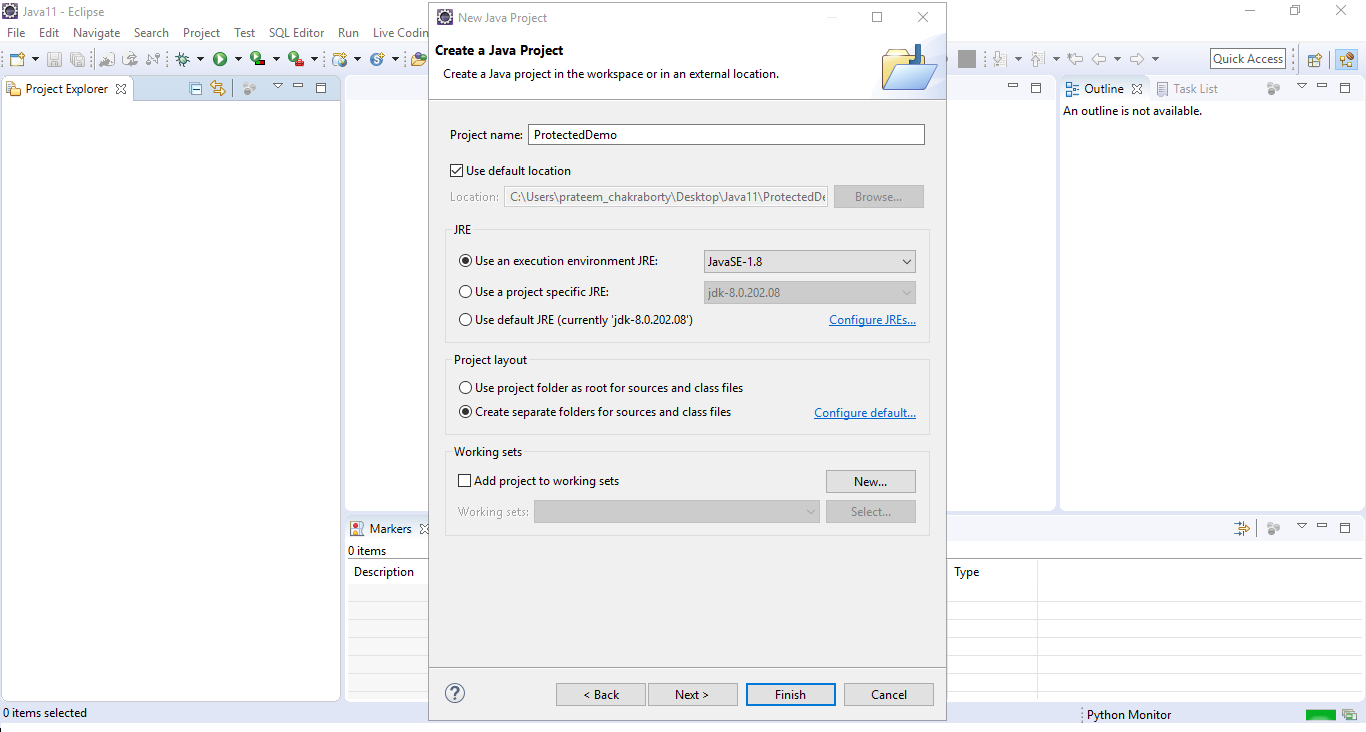
The protected access modifier can be applied only to members of a class and cannot be applied to a class unlike public and default.

Any member with protected access modifier can be accessible in the same package and in the child classes of a different package.

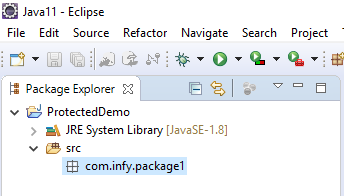
You will next understand protected access modifier better through a demo.

Please follow the steps given below to understand protected access modifier in Java.

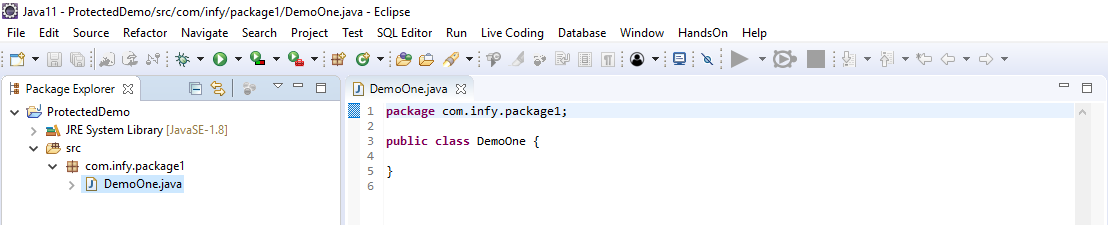
**Step 1**: Create a project named ProtectedDemo as shown below.



**Step 2**: Create a new package named com.infy.package1 as shown below.



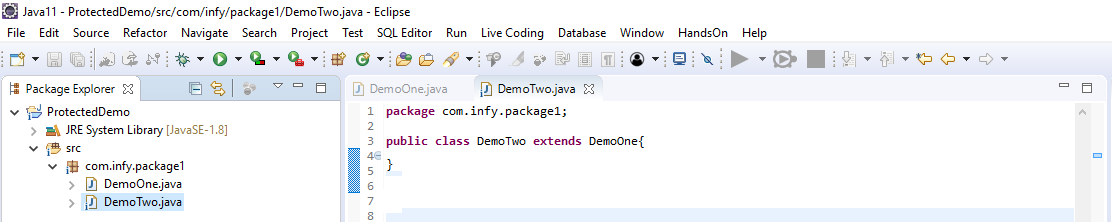
**Step 3**: Create a new class named DemoOne inside com.infy.package1 as shown below.



**Step 4**: Create a protected variable named variableOne in the class DemoOne.

1. public class DemoOne {
2. protected int variableOne;
3. }

**Step 5**: Add a child class of DemoOne named DemoTwo in com.infy.package1 as shown below.



**Step 6**: Create a method checkAccessibility as shown below in the class DemoTwo. This method will be used to check the accessibility of variableOne of class DemoOne in class DemoTwo.

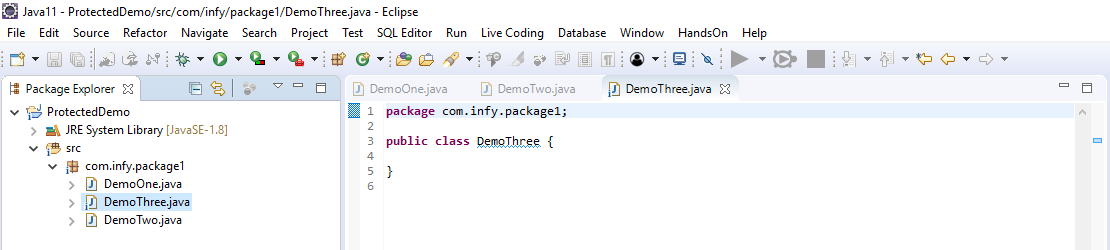
1. public class DemoTwo extends DemoOne{
2. public void checkAccessibility() {
3. }
4. }

**Step 7**: Try to access and initialize variableOne in the method checkAccessibility() as shown below.

1. public class DemoTwo extends DemoOne{
2. public void checkAccessibility() {
3. variableOne=11;
4. }
5. }

You can observe that variableOne of class DemoOne is accessible in the class DemoTwo.

**Step 8**: Add another class named DemoThree in com.infy.package1 as shown below.



**Step 9**: Create a method checkAccessibility as shown below in the class DemoThree. This method will be used to check the accessibility of variableOne of class DemoOne in class DemoThree.

1. public class DemoThree{
2. public void checkAccessibility() {
3. }
4. }

**Step 10**: Since DemoThree is not a child class of DemoOne, create an object of class DemoOne in the method checkAccessibility as shown below.

1. public class DemoThree{
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. }
5. }

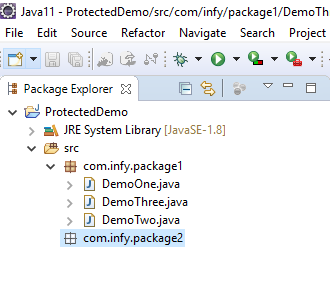
**Step 11**: Try to access and initialize variableOne in the method checkAccessibility() as shown below.

1. public class DemoThree{
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=11;
5. }
6. }

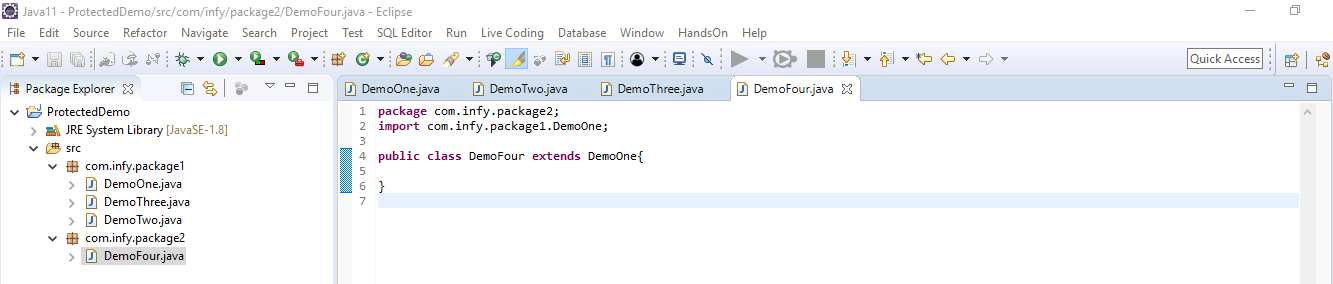
You can observe that variableOne of class DemoOne is accessible in the class DemoThree also.

This means that **a variable with protected access modifier can be accessed anywhere within the same package in which it is defined**.

**Step 12**: Add another package named com.infy.package2 in the same project as shown below.



**Step 13**: Create a child class of DemoOne of com.infy.package1 named DemoFour in com.infy.package2 as shown below.



**Step 14**: Create a method checkAccessibility as shown below in the class DemoFour. This method will be used to check the accessibility of variableOne of class DemoOne in class DemoFour.

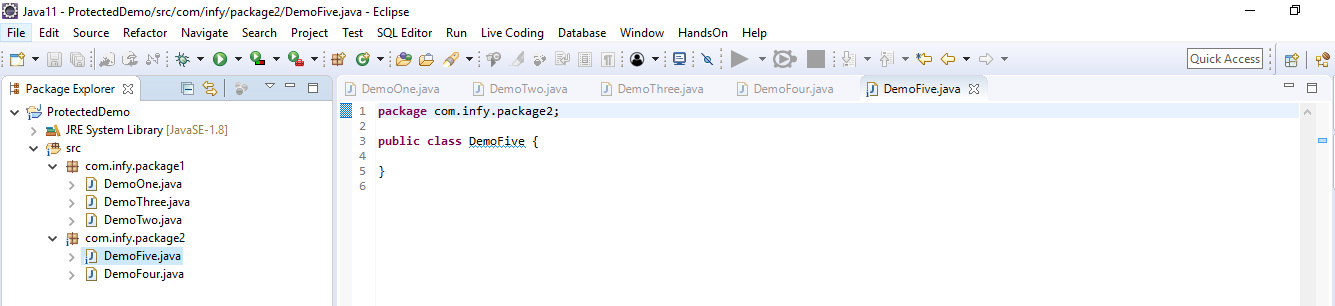
1. public class DemoFour extends DemoOne{
2. public void checkAccessibility() {
3. }
4. }

**Step 15**: Try to access and initialize variableOne in the method checkAccessibility() as shown below.

1. public class DemoFour extends DemoOne{
2. public void checkAccessibility() {
3. variableOne=11;
4. }
5. }

You can observe that variableOne of class DemoOne is accessible in the class DemoFour.

**Step 16**: Add another class named DemoFive in com.infy.package2 as shown below.



**Step 17**: Import the class named DemoOne from com.infy.package1 to class DemoFive of com.infy.package2 using the import statement as shown below.

1. package com.infy.package2;
2. import com.infy.package1.DemoOne;
3. public class DemoFive{
4. }

**Step 18**: Create a method checkAccessibility as shown below in the class DemoFive. This method will be used to check the accessibility of variableOne of class DemoOne in class DemoFive.

1. public class DemoFive{
2. public void checkAccessibility() {
3. }
4. }

**Step 19**: Since DemoFive is not a child class of DemoOne, create an object of class DemoOne in the method checkAccessibility as shown below.

1. public class DemoFive{
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. }
5. }

**Step 20**: Try to access and initialize variableOne in the method checkAccessibility() as shown below.

1. public class DemoFive{
2. public void checkAccessibility() {
3. DemoOne demoOne=new DemoOne();
4. demoOne.variableOne=11;
5. }
6. }

You can observe that you get a compilation error stating 'The field DemoOne.variableOne is not visible' while trying to access variableOne of class DemoOne in the class DemoFive as variableOne is protected and hence, can be accessed only within the same package in which it is defined or in child classes of a different package.

So, through this demo, you would have understood the scope of protected access modifier, i.e., **any member with protected access modifier can be accessible in the same package and in the child classes of a different package**.

Method Overriding:

We have seen how an object can have multiple versions of the same behavior, leading to static polymorphism. Sometimes, objects have a different way of doing things their parents do, i.e. child objects have a different implementation of the inherited behaviors.

This leads to a kind of polymorphism where a parent type appears to exhibit different implementations of the same behavior **depending upon the child type.**  
For example, an animal (parent type) can eat (behavior) by chewing (implementation), if it is a tiger (child type). Or it can eat by pecking if it is a sparrow. Or even by swallowing, if it is a python.

This display of polymorphism is **dynamic** in nature and depends upon the kind of child.

In Java, this can be achieved by Method Overriding.

Method Overriding lets you redefine a method in the child class which is already present in the parent class.

Consider the class Automobile. This class has a start method, but when it comes to the specialization of this class into Car and Bike, each of these will have a different implementation for the same.

Thus, this behavior can be redefined, or we can say, overridden in the child classes.

1. public class Automobile { *// Parent class*
2. private String vehicleNo;
3. public void start() {
4. System.out.println("Automobile starts...");
5. }
6. }

* When we override a method in the child class, it should have the same signature as that of the parent class.
* It should not have a weaker access privilege.
* Private methods cannot overridden.

Observe the child class Car, given below. It has a method, start(), which is similar to one in the parent class, Automobile.

1. public class Car extends Automobile { *// Child class*
3. @Override
4. public void start() {
5. System.out.println("Car starts...");
6. }
7. }

The @Override annotation indicates that the sub-class method is overriding the base class method. When we use this annotation, it tells the compiler that we are overriding the base class method.

In the above code, start() is overriding the method (start()) in base class.

Take a look at some main method implementations:

1. public static void main(String[] args) {
2. Automobile car = new Automobile();
3. car.start(); *// calls the start() method of the parent*
4. }
5. public static void main(String[] args) {
6. Car nano = new Car();
7. nano.start(); *// calls the start() method of the child*
8. }

# Java Annotations

 Annotations provide supplement information about the program. It starts with '@' and does not change the flow of the program.

Annotations are used to associate metadata with the program elements i.e constructors, methods, etc. It can change the perspective of the compiler for executing the program.

@Override is a built-in annotation. To learn more about annotations click [here](https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html).

Consider the code given below

1. class Student {
3. public void display() {
4. System.out.println("Student");
5. }
6. }
8. class UndergraduateStudent extends Student{
9. *//This method overrides display() of parent Student*
10. @Override
11. public void display() {
12. System.out.println("This Student is Undergraduate");
13. }
14. }
15. class GraduateStudent extends Student{
16. *//This method overrides display() of parent Student*
17. @Override
18. public void display() {
19. System.out.println("This Student is Graduate");
20. }
21. }
22. public class Tester{
23. public static void main(String[] args) {
25. Student student = new Student();
26. student.display();
28. Student obj1 = new GraduateStudent();
29. obj1.display();
31. Student obj2 = new UndergraduateStudent();
32. obj2.display();
33. }
34. }

Output :

1. Student
2. This Student is Graduate
3. This Student is Undergraduate

Note: If an object of a Student(parent) class is used to invoke the display method, then the version in the Student class will be executed, but if an object of the GraduateStudent or UndergraduateStudent (child) class is used to invoke the method, then the version in the child class will be executed.

* Rules of Method Overriding
  + Final methods cannot be overridden.
  + Access level cannot be more restrictive than that of the overridden method.
  + Static methods cannot be overridden.
  + Private methods cannot be overridden.
  + The overriding method must have the same return type and the same number of arguments as in the overridden method.
    - In the example given, display() in Student is overriden method and display() in child class is overriding method.

Observe the code given below.

1. public static void main(String[] args) {
2. Automobile car = new Car();
3. car.start(); *// Line 2*
4. }

Which method will 'line 2' invoke?

A parent class reference can refer to a child class object, i.e. child objects can substitute parent objects (Liskov substitution principle).

The version of the method that will be called is determined by the object, and this decision is taken at runtime. This is called Dynamic binding.

Besides the inherited methods, only the overridden methods can be called using the parent class reference. Any new method created in the child class will not be accessible.

Object Class:

All objects have something in common - something inherent that classifies them as objects.

In Java, the Object class is the implicit superclass of all classes.  
The methods from the Object class are inherited and can be overridden.

Some of the important methods that are present in the Object class are:

* equals()
* hashCode()
* toString()

You will now understand these methods in detail.

**equals()**compares objects to check for equality.

* By default, it uses memory addresses to compare objects for equality (just like ==).
* To make it work for different requirements, it needs to be overridden.

1. public boolean equals(Object obj) {
2. *//code to be implemented*
3. }

**hashCode()** uses an object's data to generate a hash value, which is a 32-bit integer.

* By default, it derives the hash value based on the memory address of the object being used
* In Java, hash tables and other such data structures make use of hash values for better performance
* To make it work for different requirements, it needs to be overridden

1. public int hashCode() {
2. *// code to be implemented*
3. }

For the proper usage of hash-based structures and algorithms, the hash values of equal objects should be the same. Since we have different methods for equality check and hash code generation, this calls for a contract.

**Equal objects must have equal hash codes**

In other words, if two objects are equal according to the **equals()**method, then the **hashCode()** method on each of the two objects must produce the same integer result.

**To maintain this contract both the methods should be overridden in tandem**

It is important to understand that if the hash codes of two objects are the same, it doesn't prove that the objects are equal, i.e. it is possible for two unequal objects to have the same hash codes.

**toString()** returns a textual representation of the object.

The returned text should be concise, easy to read, and informative.

By default, it returns a string consisting of the name of the object's class, the '@' character, and the unsigned hexadecimal representation of the hash code of the object.

It should be overridden to provide meaningful textual representation.

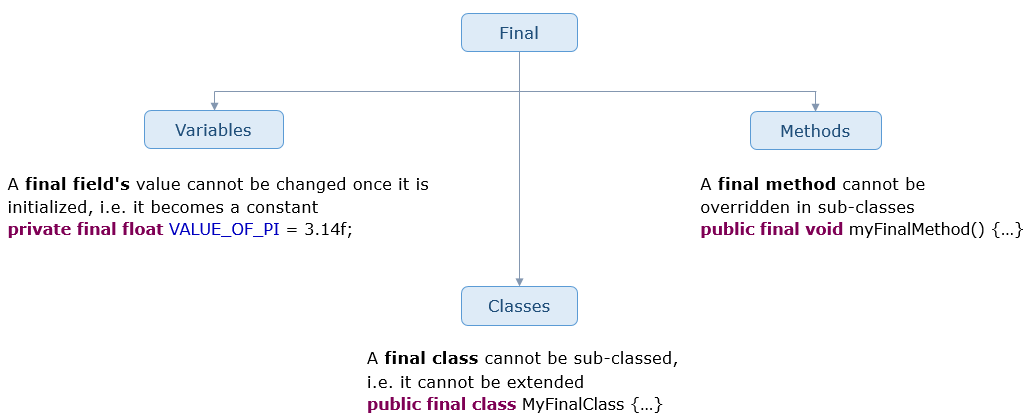
1. public String toString() {
2. *//code to be implemented*
3. }

Introduction To final:

Remember how we used π (Pi) to calculate surface areas and volumes? It is a constant with the value 3.14.

While programming, you may come across various situations where you would have to create components which should never change, i.e. remain constant. This is where the final keyword comes into picture.

This keyword can be used with variables, classes and methods.

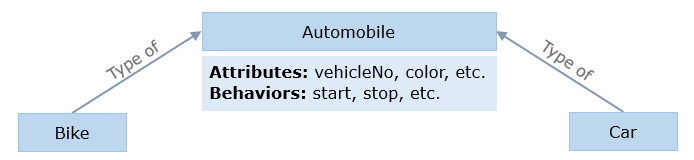


Note: Notice the identifier naming convention for final fields.

Abstrct:

Having learnt about relationships and how to represent them in Java code, let's move forward to see what our options are when we do not have the complete idea of the implementation of a class… or when we just want to specify the behaviors without providing the implementation.

Let's come back to the example of automobiles to see where such a need would arise.



All automobiles start, stop and move, but may not do so in the same way. For example, both the bike and the car can stop, but the braking mechanism of a bike is not exactly the same as that of a car. So this implementation cannot be defined in the Automobile class.

In such a case, when we do not have the complete implementation of behaviors or classes, we use the **abstract** keyword.

The abstract keyword signifies that something is not complete. It can be used with classes and methods.

An **abstract class** is a class that is incomplete. It **cannot** be instantiated.  
For it to be used, an abstract class has to be made complete by extending it.

* It is generally used when we want to have some behavior but are not sure how exactly it should be implemented.
* An abstract class encapsulates the common behaviors of all its subclasses with the help of abstract methods.
* Concrete (non-abstract) subclasses which extend an abstract class must implement all the abstract methods. Otherwise, they should be made abstract as well.

1. abstract class Automobile {
2. *// Rest of the code*
3. }

An **abstract method** is a method without any definition, i.e. the body.

* The signature of an abstract method must be preceded by the abstract keyword.
* If a class contains at least one abstract method, the class should be abstract.

A class can be abstract even without any abstract methods.

1. abstract class Automobile {
2. public abstract void start(); *// Notice the semicolons in the method declarations*
3. public abstract void stop();
4. }

 Method declarations in the abstract parent class:

1. public abstract class Automobile {
2. private String vehicleNo;
3. public abstract void start();
4. public abstract void stop();
5. public String getVehicleNo() {
6. return vehicleNo;
7. }
8. public void setVehicleNo(String vehicleNo) {
9. this.vehicleNo = vehicleNo;
10. }
11. }

Overridden methods in the child classes:

1. public class Car extends Automobile {
2. public void start() {
3. System.out.println("Car " + getVehicleNo() + " has started");
4. }
5. public void stop() {
6. System.out.println("Car " + getVehicleNo() + " has stopped");
7. }
8. }
9. public class Bike extends Automobile {
10. public void start() {
11. System.out.println("Bike " + getVehicleNo() + " has started");
12. }
13. public void stop() {
14. System.out.println("Bike " + getVehicleNo() + " has stopped");
15. }
16. }
17. public class AbstractTester {
18. public static void main(String[] args) {
19. Automobile skyline = new Car();
20. skyline.setVehicleNo("WB-4546-34");
21. Automobile ninja = new Bike();
22. ninja.setVehicleNo("KA-8675-72");
23. startAutomobile(skyline);
24. startAutomobile(ninja);
25. }
27. public static void startAutomobile(Automobile automobile) {
28. automobile.start();
29. }
30. }

**Abstract classes** enforce **inheritance**

                            and

**Abstract methods** enforce **overriding**  
  
  Hence, we achieve **dynamic binding.**

Introduction To Interface:

Let us consider the usual transport scenario. To travel from one place to the other, a taxicab has become a popular public transport medium. But doesn't taxicab belong to the Car family? So does it mean that all cars can be used for public transport? No, only a certain number of these cars can be used for public transport. Similarly, a bike taxi, bus, airplane, train, etc, all can be used for public transport. But does it mean that any bike or an airplane can be used for public transport? Again, no. There is a small portion of these that can be used.

So how can something belong to two separate families? Well, in Java, such a scenario can be implemented using the Interface. The interface in Java provides additional behaviour to a class. Hence, a taxicab along with being a car can act as a public transport using an interface.

An interface is used to define a generic template that can be implemented by various classes. It contains method signatures and constant declarations. Since Java 9, it can also have **private, default and static methods**. The methods declared in an interface are **implicitly public and abstract**, and the data fields are **implicitly public, static and final**, i.e. **constants**. An Interface specifies the alter-ego functionalities of many classes. To create an interface in Java, we have to use the **interface**keyword as shown below.

1. public interface PublicTransport{
2. double RATE = 12.0; *// Rate per km*
3. double MIN\_AMOUNT = 30.0;
4. double calculateFare();
5. }

The **implements**keyword is used to implement an interface. The classes implementing an interface must implement all the specified methods. Otherwise, they should be made abstract.

1. public class InterfaceTester {
2. public static void main(String[] args) {
3. TaxiCab olaCar = new TaxiCab();
4. *// After travelling*
5. System.out.println("Your bill amount is: Rs." + olaCar.calculateFare());
6. }
7. }
8. public class Car {
9. String brand;
10. String color;
11. String model;
12. }
13. public class TaxiCab extends Car implements PublicTransport{
14. public double calculateFare() { *// Implementing interface's method*
15. double billAmount = MIN\_AMOUNT + (RATE \* getTravelDistance());
16. return billAmount;
17. }
18. public double getTravelDistance() {
19. *// Calculates and returns the distance traveled*
20. }
21. }

An interface can extend more than one interface, and a class can implement more than one interface. This can be used to simulate **multiple inheritance** in Java.   
A class can extend from another class and at the same time implement any number of interfaces. Till Java 7, interfaces were allowed to have only empty methods or methods with no implementation. But from Java 8, interfaces are allowed to have methods with implementation.

Java now allows default and static keywords to declare interface methods.

**default** is a keyword which when attached to an interface method, allows us to provide an implementation for that method. With this, we can add additional methods in any interface without disturbing the classes which have already implemented the interface. This method can be overridden in the implementing classes if need be.

**static** is another keyword introduced which allow us to provide an implementation to the method it is attached to. The difference between *static*and *default*is that a method having *default*keyword can be overridden, whereas, one having *static*cannot.

1. public interface PassengerCar {
2. int PRICE\_PER\_KM = 30;
3. default double calculateFare(double distanceTravelled) {
4. return PRICE\_PER\_KM\*distanceTravelled;
5. }
6. static double convertToDollars(double rupees) {
7. return rupees/70;
8. }
9. }

**Note:** If a class implements multiple interfaces having the same default methods, it has to override them.  
From the overridden methods, the default methods of a specified interface can be called using the super keyword.  
E.g. PassengerCar.super.calculateFare();

Consider a Pet Adoption Center having multiple pets ranging from cats to dogs to snakes, etc. Now if we have to implement such a center in Java, there will be a method to adopt any pet. If we have to adopt a cat, then we will need a function that will help us adopt a cat, and if we want a dog, it will help us adopt a dog,

1. class PetAdoption {
2. public void adoptPet(Cat catDetails){
3. *//Assume Cat is a Java class.*
4. }
5. public void adoptPet(Dog dogDetails){
6. *//Assume Dog is a Java class.*
7. }
8. }

And this will go on for each category of pets. So if there are 50 categories, we will need 50 overloaded methods of *adoptPet()*. Can we simplify this?

Yes we can. Since a dog or a cat has to have an additional role of a Pet, we can create an Interface, Pet to describe this additional function, and make the Cat and Dog class implement the interface.

1. interface Pet {
2. *//Different methods*
3. }
4. class Cat implements Pet {
5. String breed;
6. }
7. class Dog implements Pet {
8. String furColour;
9. }

Since Dog and Cat are implementing the interface Pet, they will get the extra behaviour of a Pet. With this, we can call the dog or the cat as Pets. And with that we can implement objects like given below,

1. Pet pet1 = new Cat();
2. Pet pet2 = new Dog();

Due to this, we can modify the PetAdoption class to have just one method, which will accept Pet Details for adoption,

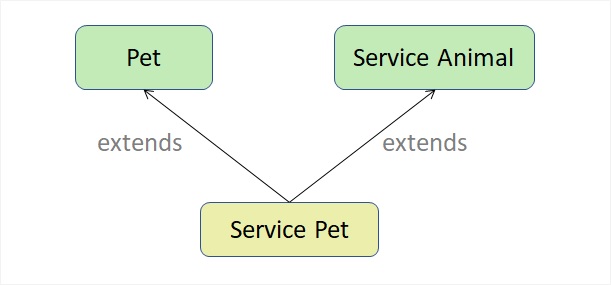
1. class PetAdoption {
2. public void adoptPet(Pet petDetails){
3. *//Method implementation to adopt a pet.*
4. }
5. }

So, instead of creating a new method for each Pet category, we can have only one method, which accepts a Pet object, which can still take any object that implements the Pet interface. Similarly, if we have any more classes which are Pets, we need not create new functions for each class. We can just make the class implement Pet, hence allowing that class to be recognized as a Pet.

Unlike inheritance in classes, inheritance in interface allows extending of one or more interfaces. And when this is implemented, all the functions of the Base Interface is transferred to the Derived Interface.

Consider a scenario of guinea pigs. These animals are classic pet materials. You can take them as your pet and play with them all day. Similarly, if we consider a horse. Very expensive to be a pet, but very useful to be serving in the army. A guinea pig is a misfit in the army, just like how a horse in the present time, is a misfit to be a pet. But what if we have an animal that was both?

Consider a scenario of a dog. The dog belongs to the Animal class, but can also be a pet. Some dogs can also be service animals used in the military. And some of the dogs can be both pets and service animals. How will this scenario be represented/implemented in Java? How will a dog, who is associated with the animal class be assigned two extra behavioral characteristics? This can be done using inheritance in the interface.

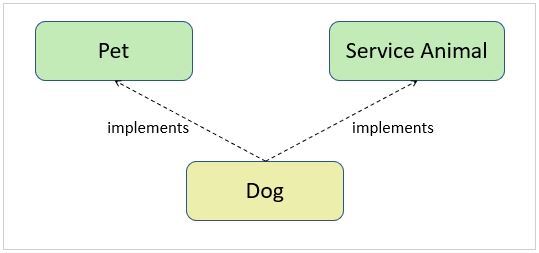


Let us see the implementation of this Java,

1. interface Pet {
2. *//Method definitions describing a pet*
3. }
4. interface ServiceAnimal {
5. *//Method definitions describing service animal*
6. }
7. interface ServicePet extends Pet, ServiceAnimal{
8. *//This will have methods from Pet and Service Animal interfaces along with the methods of its own*
9. }

As you can see in the code, multiple inheritance of interfaces is permitted. And any class implementing the ServicePet interface will get the attributes of both Pet and Service Animal.

Similarly, even a class can implement two different interfaces at the same time,



As you can see above, a Dog class is implementing (dotted lines indicate *implements*and solid lines indicate *extends*) both, Pet and Service Animal Interface. With this, the Dog class will get the attributes and methods from Pet and Service Animal interface. The implementation of the above shown image can be,

1. interface Pet {
2. *//Method definitions describing a pet*
3. }
4. interface ServiceAnimal {
5. *//Method definitions describing service animal*
6. }
7. class Dog implements Pet, ServiceAnimal{
8. *//This will have methods from Pet and Service Animal interfaces along with the methods and attributes of its own*
9. }

Functional Interface:

Java 8 introduces a new category of Interface called **Functional Interface**. They are special cases of regular interfaces that have only one abstract method and one or more default methods.

1. interface Calculator {
2. int doOperation(int num1, int num2);
3. default boolean checkValues(int num1, int num2) {
4. return (num1 > 0 && num2 > 0);
5. }
6. }

The above interface is a functional interface. But by chance, if we add one or more abstract methods in this interface compiler will not complain. The compiler will simply consider this as any normal interface.

In order for the compiler to restrict the addition of more abstract methods, we need to use annotation **@FunctionalInterface**.

1. @FunctionalInterface
2. interface Calculator{
3. int operation(int num1, int num2);
4. }

Now, if by chance we add another abstract method compiler will show an error.

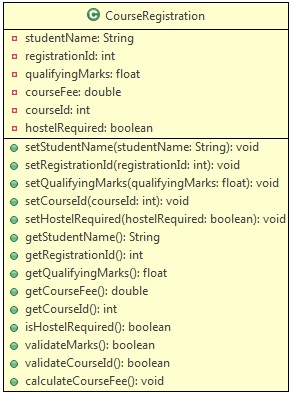
Functional Interfaces are very useful for defining Lambda Expressions which we will study later in this course.

There are many inbuilt functional interfaces given by Java.

**Problem Description:**

My University wants to automate the process of registering students to different courses. The CourseRegistration class has been created to help in the process of course registration. The class diagram is given below:

**CourseRegistration:**



A student is eligible to get admission only if the qualifying marks is in the range of 65 to 100(both inclusive). At present, the university offers five courses with course ids in the range 1001 to 1005(both inclusive). Based on the marks in the qualifying exam, a discount is provided in the course fee. Use the table below to determine the discount:

| **Discount** | |
| --- | --- |
| **Marks Range** | **Discount** |
| **65-69** | **5%** |
| **70-84** | **10%** |
| **>=85** | **15%** |

Use the table below to get the base fee of different courses:

| **Base fee** | |
| --- | --- |
| **Course Id** | **Fee** |
| **1001** | **Rs.55,000** |
| **1002** | **Rs.35,675** |
| **1003** | **Rs.28,300** |
| **1004** | **Rs.22,350** |
| **1005** | **Rs.1,15,000** |

**Method Description:**

* **validateMarks()**: Used to validate qualifying exam marks - qualifying marks is in the range of 65 to 100(both inclusive)
* **validateCourseId()**: Used to validate the course entered, based on the courseId - given in the table above
* **calculateCourseFee()**: Used to calculate the course fee after applying the discount.

Use the Tester class to set the values and call appropriate methods to display the outputs for the inputs mentioned below:

**Sample:​​​​​​​**

| **Input 1** | |
| --- | --- |
| **Attributes** | **Values** |
| **Name** | **Peter** |
| **Registration Id** | **5001** |
| **Qualifying Exam Marks** | **58** |
| **Course Id** | **1005** |
| **Hostel Required** | **true** |

**Output:**

**Marks is less than 65. You are not eligible for admission!!**

| **Input 2** | |
| --- | --- |
| **Attributes** | **Values** |
| **Name** | **Peter** |
| **Registration Id** | **5001** |
| **Qualifying Exam Marks** | **68** |
| **Course Id** | **1006** |
| **Hostel Required** | **true** |

**Output:**

**Invalid Course Id. Please try again!!**​​​​​​​

| **Input 3** | |
| --- | --- |
| **Attributes** | **Values** |
| **Name** | **Peter** |
| **Registration Id** | **5001** |
| **Qualifying Exam Marks** | **78** |
| **Course Id** | **1005** |
| **Hostel Required** | **false** |

**Output:**

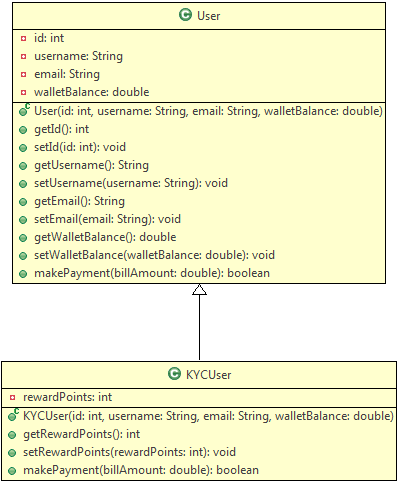
**\*\*\*\*\*\*\*\*Course Allocation Details\*\*\*\*\*\*\*\*  
                                                           Student Name                        :Peter  
                                                           Course Id                               :1005  
                                                           Qualifying Exam Marks        :78.0  
                                                           Student's Registration Id     :5001  
                                                           Total course fee                    :103500.0  
​​​​​​​                                                           Hostel Required                    : No**

Create a Java Project and implement the given requirements.

**Note:** Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

**Problem Description:  
EPay Wallet is a wallet application using which its users can pay various bills. Users can make payments only if they have enough wallet balance.**

There are two kinds of users as illustrated by the class diagrams below. General users can make regular payments, whereas KYC users get reward points for every payment.



**Method Description:**

**User:**

**makePayment(double billAmount):** This method makes payment by deducting the bill amount from wallet if there is enough balance.

* If the balance is not enough, it returns false
* If the balance is sufficient, it deducts the specified bill amount from the wallet and returns true

**KYCUser:**

**makePayment(double billAmount):** This method overrides the parent method to make payment as well as to credit reward points to the user.

* It uses the payment functionality of the parent class
* If payment is successful, it adds 10% of the bill amount as reward points
* It returns true or false depending on whether the payment was successful or not

The primary EPay Wallet operations (currently only one) are to be defined in a separate class as follows:

This image shows the class diagram for class EPayWallet. It has one static method processPaymentByUser(user: User, billAmount: double): void

**Method Description:**

**processPaymentByUser(User user, double billAmount):** This is a static method to process the bill payment by any EPay Wallet user.

* It uses the makePayment() method of the user to process payments, and displays success or error messages depending on whether the payment was successful or not.
* It shows the wallet balance of the user
* If the user is a KYC user, it shows the reward points as well

**Note:**

* Have a look at the sample output to understand the messages to be displayed
* You can use Java's instanceOf operator to check the type of an object

Use the Tester class to test the above functionalities. Create User and KYCUser objects with different values and call the processPaymentByUser() method of EPayWallet class to process payments.

**Sample Input:​​​​​​​**

|  |  |
| --- | --- |
| **User** | |
| **id** | **101** |
| **username** | **Jack** |
| **email** | **jack@infy.com** |
| **walletBalance** | **1000** |

|  |  |
| --- | --- |
| **KYC User** | |
| **id** | **201** |
| **username** | **Jill** |
| **email** | **jill@infy.com** |
| **walletBalance** | **3000** |

|  |  |
| --- | --- |
| **Sample payments** | |
| **Jack** | **700** |
| **Jill** | **1500** |
| **Jill** | **800** |
| **Jill** | **1200** |

Sample Output:

1. Congratulations Jack, payment of 700.0 was successful
2. Your wallet balance is 300.0
3. --------------------------------------------
4. Congratulations Jill, payment of 1500.0 was successful
5. Your wallet balance is 1500.0
6. You have 150 reward points
7. --------------------------------------------
8. Congratulations Jill, payment of 800.0 was successful
9. Your wallet balance is 700.0
10. You have 230 reward points
11. --------------------------------------------
12. Sorry Jill, not enough balance to make payment
13. Your wallet balance is 700.0
14. You have 230 reward points
15. --------------------------------------------

Create a new Java project to implement the requirements given above.

**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

**Problem Description:**

RRTechnicals, a study center allows students to register for two types of certification courses - CrashCourseCertification and RegularCourseCertification. For this, the center has to comply with the standards set for all certifications.

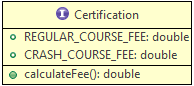
Also, an admission test is conducted for all the students who want to apply, and the marks scored in the admission test is considered for a discount on their total fees as per the table below:

| **Discount table** | |
| --- | --- |
| **Marks** | **Discount %** |
| **90 or greater** | **10** |
| **75 to 89 (Both inclusive)** | **5** |

An extra 12.36% service tax is applicable on discounted fee for crash course. Since regular course students are paying more amount, they are exempted from the service tax.

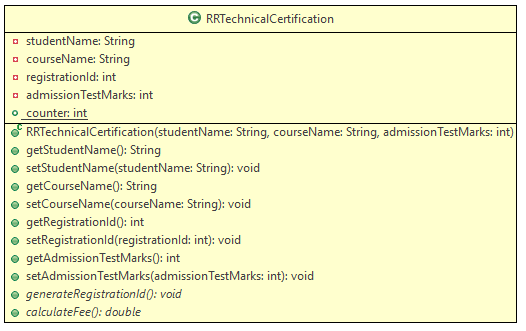
Create the required interface and classes as per class diagrams given below:

**Certification:**

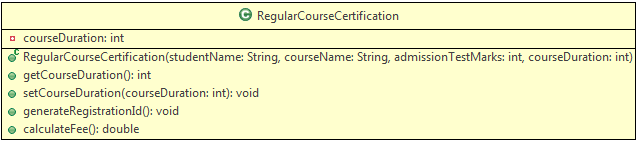


The **Certification**interface provides the standards for all certifications. It specifies the standard fees for the courses and a method to calculate the final fee for the study centers. The regular courses can be of 5-6 months duration for which the fee is Rs. 2000 per month. The crash course duration is fixed to 2 months and the fee is Rs. 5000.

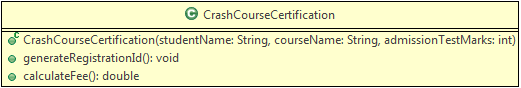
**RRTechnicalCertification: The parent class which complies with the certification interface and provides the common features for different types of RRTechnical certifications.**



**RegularCourseCertification:**



**CrashCourseCertification:**



**Method Description:**

**generateRegistrationId():** This method is used to generate a unique registration id for the certification of each student. The id for regular courses must contain a 4 digit odd number, starting with 1001. For crash course, the id must contain a 4 digit even number.  
**Note:** The registration id should get generated as soon as a student is registered (i.e. when an object is created) and it need not follow any sequence.

**calculateFee()**: This method is used to calculate the total certification fees for the students of RRTechnicals. Make appropriate use of the interface and the above mentioned business conditions to determine the total fee amount.

Use the Tester class to check your code and display the results as shown in the sample output.

**Sample:**

| **Input (for RegularCourseCertification)** | |
| --- | --- |
| **Attribute** | **Value** |
| **Student Name** | **Rakesh** |
| **Course Name** | **J2EE** |
| **Course Duration** | **5** |
| **Admission Test Marks** | **85** |

**Output:**

1. Regular Course Registration Details
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. Student Name : Rakesh
4. Course Name : J2EE
5. Course Duration : 5 months
6. Registration ID : 1001
7. Fees : 9500.0

| **Input (for CrashCourseCertification)** | |
| --- | --- |
| **Attribute** | **Value** |
| **Student Name** | **Roshan** |
| **Course Name** | **Angular** |
| **Admission Test Marks** | **71** |

**Output:**

1. Crash Course Registration Details
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. Student Name : Roshan
4. Course Name : Angular
5. Course Duration : 2 months
6. Registration ID : 1002
7. Fees : 5618.0

Create a new Java Project and implement the given requirements.

**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

**Academy**is an online academy which invites candidates to participate in various events, and also provides grades for each candidate based on his/her performance. They need your help in two modules,

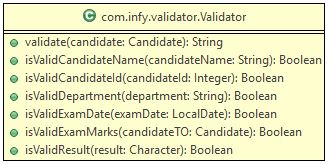
* Adding candidates
* Getting candidate reports

| ****Artifacts**** | |
| --- | --- |
| **File Name** | **Status** |
| **com.infy.dao.CandidateDAO.java** | **Implemented** |
| **com.infy.dao.CandidateDAOImpl.java** |
| **com.infy.model.Candidate.java** |
| **com.infy.model.CandidateReport.java** |
| **com.infy.service.CandidateService.java** |
| **com.infy.userinterface.CandidateTester.java** |
| **com.infy.service.CandidateServiceImpl.java** | **To be modified** |
| **com.infy.validator.Validator.java** | **To be implemented** |
| **com.infy.test.CandidateServiceTest.java** | **To be ignored for now** |

[Download the starter code](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_01297182771365478478_shared/web-hosted/assets/InfyAcademyDay2ToTrainee1641474276703.zip) for this assignment. Refer **Appendix->Import Maven project in Eclipse**at the end of the course for importing the project and updating to Maven**.**

**Validator:**

This class validates the inputs provided by the user.



**validate(Candidate candidate):**

This method should call the validation methods for individual inputs, and if any method returns false, then the String value corresponding the to input being validated based on the table given below, will be returned,

| ****Exception messages**** | |
| --- | --- |
| **Input** | **Exception message** |
| **candidateld** | **The entered candidate ID is invalid.** |
| **candidateName** | **The entered candidate name is invalid.** |
| **mark1, mark2, mark3** | **The entered exam marks are invalid.** |
| **result** | **The entered result is invalid.** |
| **department** | **The entered Department name is invalid.** |
| **examDate** | **The entered Exam Date is invalid.** |

**isValidCandidateName(String candidateName):**

* This method validates the received **candidateName**.
* The **candidateName**contain only the letters of the English alphabet.
* It should not contain any spaces or special characters.
* If any of the above conditions are not satisfied, then return false, else return true.

**isValidCandidateId(Integer candidateId):**

* This method validates the received **candidateId**.
* The **candidateId**should have only 5 digits.
* If the above condition is not satisfied, then return false, else return true.

**isValidDepartment(String department):**

* This method validates the received **candidateId**.
* The **department**can be one among, **ECE**, **CSE**, **IT**, **EEE**.
* If the above condition is not satisfied, then return false, else return true.

**isValidExamDate(LocalDate examDate):**

* This method validates the received **examDate**.
* The **examDate**should be a past date.
* If the above condition is not satisfied, then return false, else return true.

**isValidExamMarks(Candidate candidateTO):**

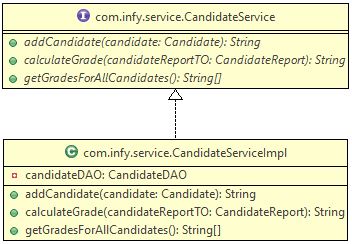
* This method validates the mark attributes (**mark1**, **mark2**and **mark3**) in the Candidate object received.
* None of the mark attributes should have a negative value.
* If the above condition is not satisfied, then return false, else return true.

**isValidResult(Character result):**

* This method validates the received **result**.
* The result should be either **P** or **F**.
* If the above condition is not satisfied, then return false, else return true.

**CandidateServiceImpl:**

This class is already implemented. As shown below, it implements an interface CandidateService. You have to make a few modifications for this class based on the class diagram given below,



* Create an instance variable as shown in the class diagram and give it the object of CandidateDAOImpl class.
* Use this instance variable instead of the local variables used in the class while invoking CandidateDAOImpl methods.

**addCandidate(Candidate candidate):**

* Before the marks and result verification, invoke the validate() method of the Validator class by passing the candidate object as the parameter, which returns a String value.
* If the String value returned is not null, then return the same String value to the CandidateTester class.
* Else, let the process present in the class continue.
* Make sure the variable used to invoke CandidateDAOImpl methods is the instance variable created in the previous steps.

**getGradesForAllCandidates():**

* Make sure the variable used to invoke CandidateDAOImpl methods is the instance variable created in the previous steps.

### ****Sample Input and Output:****

### ****Add Candidate****

| ****Input 1**** | |
| --- | --- |
| **Attribute** | **Value** |
| **candidateld** | **12346** |
| **candidateName** | **Sam** |
| **mark1** | **51** |
| **mark2** | **56** |
| **mark3** | **78** |
| **result** | **P** |
| **department** | **ECE** |
| **examDate** | **29th May, 2014** |

### ****Output:****

* **Invalid Data 1:**For invalid candidateName, **Sam123**, the output should be
* **Invalid Data 2**: For invalid candidateId, **1234**, the output should be,
* **Invalid Data 3:** For invalid department, **EC**, the output should be,
* **Invalid Data 4:** For invalid examDate, **29th March, 20140**, the output should be,
* **Invalid Data 5:** For invalid examMarks, **mark1 = 12**, **mark2=0 , mark3=67** the output should be,
* **Invalid Data 6:** For invalid result, **G**, the output should be,
* **Invalid Data 7**: For examMarks, **mark1=66**, **mark2=70**, **mark3=13,** and the result is **P**, the output should be,

### ****Get Candidates Results:****​​​​​​​

| ****Output**** | |
| --- | --- |
| **Candidate ID** | **Result** |
| **80004** | **B** |
| **80005** | **NA** |
| **80002** | **A** |
| **80001** | **NA** |
| **80003** | **C** |

**Note**: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.