***Classes, Inheritance, Exceptions, Packages and Interfaces***

***Question bank***

1. ***describe the structure of java class along with the possible member***

***with example explain the uses of this keyword***

1. ***What is inheritance? Explain how inheritance is achieved in java***
2. ***with example explain the uses of this keyword***
3. ***what is method overriding explain with example***
4. ***explain abstract class and abstract function with example***
5. ***with example explain final modifier***
6. ***What is package? Explain how to create and use packages.***
7. ***With example explain How to access package from another package***
8. ***List and explain java Access Specifiers.***
9. ***What is interface?expalin with example how to create and implement interfaces***
10. ***Write a program to illustrate the interface variables***
11. ***With example explain how to implement and extends the multiple interfaces***
12. ***What is nested interface explain with example?***
13. ***What is exception list and explain the different types of exception(Exception hierarchy)***
14. ***With example explain how exception will be handled in java***
15. ***With example explain how to create user defined exception***
16. ***What is chained exception explain with example***
17. ***Write a note on i)object class***

***ii)finalize() method***

## *1 Classes in Java*

*A class is a blueprint from which individual objects are created.*

*Following is a sample of a class.*

### *Example*

*public class Dog {*

*String breed;*

*int ageC*

*String color;*

*void barking() {*

*}*

*void hungry() {*

*}*

*void sleeping() {*

*}*

*}*

*A class can contain any of the following variable types.*

* ***Local variables****− Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.*
* ***Instance variables****− Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.*
* ***Class variables****− Class variables are variables declared within a class, outside any method, with the static keyword.*
* *A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.*

***Following are some of the important topics that need to be discussed when looking into classes of the Java Language****.*

## *Constructors*

* *When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.*
* *Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.*

*Following is an example of a constructor −*

### *Example*

*public class Puppy {*

*public Puppy() {*

*}*

*public Puppy(String name) {*

*// This constructor has one parameter, name.*

*}*

*}*

## *Creating an Object*

*As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.*

*There are three steps when creating an object from a class −*

* ***Declaration****− A variable declaration with a variable name with an object type.*
* ***Instantiation****− The 'new' keyword is used to create the object.*
* ***Initialization****− The 'new' keyword is followed by a call to a constructor. This call initializes the new object.*

*Following is an example of creating an object −*

### *Example*

*public class Puppy {*

*public Puppy(String name) {*

*// This constructor has one parameter, name.*

*System.out.println("Passed Name is :" + name );*

*}*

*public static void main(String []args) {*

*// Following statement would create an object myPuppy*

*Puppy myPuppy = new Puppy( "tommy" );*

*}*

*}*

*If we compile and run the above program, then it will produce the following result −*

### *Output*

*Passed Name is :tommy*

## *Accessing Instance Variables and Methods*

* *Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path −*

*/\* First create an object \*/*

*ObjectReference = new Constructor();*

*/\* Now call a variable as follows \*/*

*ObjectReference.variableName;*

*/\* Now you can call a class method as follows \*/*

*ObjectReference.MethodName();*

### *Example*

*This example explains how to access instance variables and methods of a class.*

*public class Puppy {*

*int puppyAge;*

*public Puppy(String name) {*

*// This constructor has one parameter, name.*

*System.out.println("Name chosen is :" + name );*

*}*

*public void setAge( int age ) {*

*puppyAge = age;*

*}*

*public int getAge( ) {*

*System.out.println("Puppy's age is :" + puppyAge );*

*return puppyAge;*

*}*

*public static void main(String []args) {*

*/\* Object creation \*/*

*Puppy myPuppy = new Puppy( "tommy" );*

*/\* Call class method to set puppy's age \*/*

*myPuppy.setAge( 2 );*

*/\* Call another class method to get puppy's age \*/*

*myPuppy.getAge( );*

*/\* You can access instance variable as follows as well \*/*

*System.out.println("Variable Value :" + myPuppy.puppyAge );*

*}*

*}*

*If we compile and run the above program, then it will produce the following result −*

### *Output*

*Name chosen is :tommy*

*Puppy's age is :2*

*Variable Value :2*

***2 this keyword:***

* *this is a keyword in java which refers to the current instance of the class. Every constructor and all non-static methods in java have this as an implicitly parameter.*
* *Note: When n numbers of parameters are passed in a method then from JRE point of view n+1 number of parameters are passed in the method. One additional parameter is this. If this keyword is used in constructor or method then this must be the first statement.*

*Use of this keyword in java:*

*1. this can be used to differentiate between instance variable and local variable.*

*2. this can be used in constructor chaining. this() and super() must be the first statement.*

*3. this can be used to invoke current class method implicitly.*

*4. this can be used to return current instance of the class.*

*5. this can be used as a parameter in constructor call.*

*6. this can be used as a parameter in method call.*

*1. this can be used to differentiate between instance variable and local variable.*

*If local variable and instance variables are same than compiler will not be able to distinguish them.*

*Example:*

*ThisExample1.java*

|  |
| --- |
| */\*\**  *\* This program is used to show that if local variable and*  *\* instance variables are same than compiler will not be able*  *\* to distinguish them.*    *\*/*  *class Student{*  *//instance variable.*  *int rollNo;*  *String name;*    *Student(String name, int rollNo){*  *//local variable.*  *name = name;*  *rollNo = rollNo;*  *}*    *public void displayDetails(){*  *System.out.println("RollNo = " + rollNo);*  *System.out.println("name = " + name);*  *}*  *}*    *public class ThisExample1 {*  *public static void main(String args[]){*  *//creating Student class object.*  *Student stu1 = new Student("jai", 6);*  *//method call*  *stu1.displayDetails();*  *}*  *}* |

*Output:*

|  |
| --- |
| *RollNo = 0*  *name = null* |

* *If the constructor modified like below*

*Student(String name, int rollNo){*

*This. name = name;*

*This.rollNo = rollNo;*

*}*

*The output will be*

*Output:*

|  |
| --- |
| *RollNo = 6*  *name = jai* |

# *3 Java - Inheritance*

* *Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.*
* *The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).*

## *extends Keyword*

* *Extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.*

*Syntax*

*class Super {*

*.....*

*.....*

*}*

*class Sub extends Super {*

*.....*

*.....*

*}*

* *Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My\_Calculation.*
* *Using extends keyword, the My\_Calculation inherits the methods addition() and Subtraction() of Calculation class.*

*Example*

*class Calculation {*

*int z;*

*public void addition(int x, int y) {*

*z = x + y;*

*System.out.println("The sum of the given numbers:"+z);*

*}*

*public void Subtraction(int x, int y) {*

*z = x - y;*

*System.out.println("The difference between the given numbers:"+z);*

*}*

*}*

*public class My\_Calculation extends Calculation {*

*public void multiplication(int x, int y) {*

*z = x \* y;*

*System.out.println("The product of the given numbers:"+z);*

*}*

*public static void main(String args[]) {*

*int a = 20, b = 10;*

*My\_Calculation demo = new My\_Calculation();*

*demo.addition(a, b);*

*demo.Subtraction(a, b);*

*demo.multiplication(a, b);*

*}*

*}*

*Compile and execute the above code as shown below.*

*javac My\_Calculation.java*

*java My\_Calculation*

*After executing the program, it will produce the following result −*

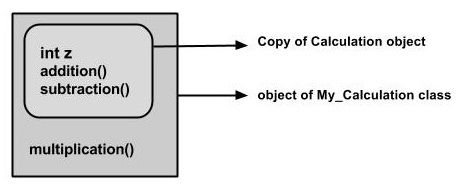
*Output*

*The sum of the given numbers:30*

*The difference between the given numbers:10*

*The product of the given numbers:200*

* *In the given program, when an object to My\_Calculation class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.*

**

***Storing child class object inside parent class reference***

* *The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.*
* *If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable ( cal in this case) you cannot call the method multiplication(), which belongs to the subclass My\_Calculation.*

*Calculation cal = new My\_Calculation();*

*demo.addition(a, b);*

*demo.Subtraction(a, b);*

***Note −*** *A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.*

## *4 The super keyword*

* *The super keyword is similar to this keyword. Following are the scenarios where the super keyword is used.*
* *It is used to differentiate the members of superclass from the members of subclass, if they have same names.*
* *It is used to invoke the superclass constructor from subclass.*

### *Differentiating the Members*

* *If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.*

*super.variable*

*super.method();*

### *Sample Code*

* *This section provides you a program that demonstrates the usage of the super keyword. In the given program, you have two classes namely Sub\_class and Super\_class, both have a method named display() with different implementations, and a variable named num with different values.*
* *We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.*

*Example*

*class Super\_class {*

*int num = 20;*

*// display method of superclass*

*public void display() {*

*System.out.println("This is the display method of superclass");*

*}*

*}*

*public class Sub\_class extends Super\_class {*

*int num = 10;*

*// display method of sub class*

*public void display() {*

*System.out.println("This is the display method of subclass");*

*}*

*public void my\_method() {*

*// Instantiating subclass*

*Sub\_class sub = new Sub\_class();*

*// Invoking the display() method of sub class*

*sub.display();*

*// Invoking the display() method of superclass*

*super.display();*

*// printing the value of variable num of subclass*

*System.out.println("value of the variable named num in sub class:"+ sub.num);*

*// printing the value of variable num of superclass*

*System.out.println("value of the variable named num in super class:"+ super.num);*

*}*

*public static void main(String args[]) {*

*Sub\_class obj = new Sub\_class();*

*obj.my\_method();*

*}*

*}*

*Output*

*This is the display method of subclass*

*This is the display method of superclass*

*value of the variable named num in sub class:10*

*value of the variable named num in super class:20*

## *Invoking Superclass Constructor*

* *If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.*

*super(values);*

### *Sample Code*

* *The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a string value, and we used the super keyword to invoke the parameterized constructor of the superclass.*

*Copy and paste the following program in a file with the name Subclass.java*

*Example*

*class Superclass {*

*int age;*

*Superclass(int age) {*

*this.age = age;*

*}*

*public void getAge() {*

*System.out.println("The value of the variable named age in super class is: " +age);*

*}*

*}*

*public class Subclass extends Superclass {*

*Subclass(int age) {*

*super(age);*

*}*

*public static void main(String argd[]) {*

*Subclass s = new Subclass(24);*

*s.getAge();*

*}*

*}*

*Compile and execute the above code using the following syntax.*

*javac Subclass*

*java Subclass*

*On executing the program, you will get the following result −*

*Output*

*The value of the variable named age in super class is: 24*

# *5 Java – Overriding(dynamic method dispatch)*

* *If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.*
* *The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.*
* *In object-oriented terms, overriding means to override the functionality of an existing method.*

### *Example*

*Let us look at an example.*

*class Animal {*

*public void move() {*

*System.out.println("Animals can move");*

*}*

*}*

*class Dog extends Animal {*

*public void move() {*

*System.out.println("Dogs can walk and run");*

*}*

*}*

*public class TestDog {*

*public static void main(String args[]) {*

*Animal a = new Animal(); // Animal reference and object*

*Animal b = new Dog(); // Animal reference but Dog object*

*a.move(); // runs the method in Animal class*

*b.move(); // runs the method in Dog class*

*}*

*}*

*This will produce the following result −*

### *Output*

*Animals can move*

*Dogs can walk and run*

* *In the above example, you can see that even though b is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.*
* *Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.*

*Consider the following example −*

### *Example*

*class Animal {*

*public void move() {*

*System.out.println("Animals can move");*

*}*

*}*

*class Dog extends Animal {*

*public void move() {*

*System.out.println("Dogs can walk and run");*

*}*

*public void bark() {*

*System.out.println("Dogs can bark");*

*}*

*}*

*public class TestDog {*

*public static void main(String args[]) {*

*Animal a = new Animal(); // Animal reference and object*

*Animal b = new Dog(); // Animal reference but Dog object*

*a.move(); // runs the method in Animal class*

*b.move(); // runs the method in Dog class*

*b.bark();*

*}*

*}*

*This will produce the following result −*

### *Output*

*TestDog.java:26: error: cannot find symbol*

*b.bark();*

*^*

*symbol: method bark()*

*location: variable b of type Animal*

*1 error*

* *This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.*

## *Rules for Method Overriding*

* *The argument list should be exactly the same as that of the overridden method.*
* *The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.*
* *The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.*
* *Instance methods can be overridden only if they are inherited by the subclass.*
* *A method declared final cannot be overridden.*
* *A method declared static cannot be overridden but can be re-declared.*
* *If a method cannot be inherited, then it cannot be overridden.*
* *A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.*
* *A subclass in a different package can only override the non-final methods declared public or protected.*
* *Constructors cannot be overridden.*

## *Using the super Keyword*

*When invoking a superclass version of an overridden method the super keyword is used.*

### *Example*

*class Animal {*

*public void move() {*

*System.out.println("Animals can move");*

*}*

*}*

*class Dog extends Animal {*

*public void move() {*

*super.move(); // invokes the super class method*

*System.out.println("Dogs can walk and run");*

*}*

*}*

*public class TestDog {*

*public static void main(String args[]) {*

*Animal b = new Dog(); // Animal reference but Dog object*

*b.move(); // runs the method in Dog class*

*}*

*}*

*This will produce the following result −*

### *Output*

*Animals can move*

*Dogs can walk and run*

# *6 Abstract Classes and function*

* *Abstraction refers to the ability to make a class abstract in OOP. An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.*
* *If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.*

## *Abstract Class*

*Use the abstract keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the class keyword. The following shows an example of how abstract class can be inherited and used.*

*import java.io.\*;*

*abstract class Person*

*{*

*int age;*

*string name;*

*int getAge()*

*{*

*return age;*

*}*

*}*

*class Employee extends Person*

*{*

*public static void main()*

*{*

*Employee emp = new Employee();*

*emp.age=34*

*emp.name = "Emp1";*

*System.out.println(emp.name);*

*System.out.println(emp.getAge());*

*}*

*}*

*When we compile and run the above program, we will get the following output.*

*Emp1*

*34*

## *Abstract functions*

* *Similar to classes, funtion can also be made abstract. The implementation of such function is not given in its class but should be provided in the class that inherits the class with abstract function.*
* *An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.*
* *Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.*
* *If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.*
* *The abstract method ends with a semicolon. Example: public void sample();*

*A above example is updated with abstract function and is given below.*

*import java.io.\*;*

*abstract class Person*

*{*

*int age;*

*string name;*

*int getAge()//abstract class can hava concrete function*

*{*

*return age;*

*}*

*}*

*void print();//abstract function*

*}*

*class Employe eextends Person*

*{*

*int empID;*

*void print()//overriding abstract function*

*{*

*System.out.println(this.name);*

*System.out.println(this.age);*

*}*

*}*

*void main()*

*{*

*Employee emp = new Employee();*

*emp.age=34*

*emp.name = "Emp1";*

*System.out.println(emp.getAge());//using parent class function*

*Emp.print();//using overridden defnition*

*}*

*When we compile and run the above program, we will get the following output.*

*34*

*Emp1*

*34*

## *7 The Final Modifier*

### *Final Variables*

* *A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object. However, the data within the object can be changed. So, the state of the object can be changed but not the reference.*
* *With variables, the final modifier often is used with static to make the constant a class variable.*

*Example*

*public class Test {*

*final int value = 10;*

*// The following are examples of declaring constants:*

*public static final int BOXWIDTH = 6;*

*static final String TITLE = "Manager";*

*public void changeValue() {*

*value = 12; // will give an error*

*}*

*}*

### *Final Methods*

* *A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.The main intention of making a method final would be that the content of the method should not be changed by any outsider.*

*Example*

*You declare methods using the final modifier in the class declaration, as in the following example −*

*public class Test {*

*public final void changeName() {*

*// body of method*

*}*

*}*

### *Final Classes*

* *The main purpose of using a class being declared as final is to prevent the class from being subclassed(inheritance). If a class is marked as final then no class can inherit any feature from the final class.*

*Example*

*public final class Test {*

*// body of class*

*}*

*Public child extends test{// error test is final cannot be extended*

*}*

# *8 Java - Packages*

* *Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.*
* *A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations ) providing access protection and namespace management.*

*Some of the existing packages in Java are −*

* *java.lang − bundles the fundamental classes*
* *java.io − classes for input , output functions are bundled in this package*
* *Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.*
* *Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.*

## *Creating a Package*

* *While creating a package, you should choose a name for the package and include a package statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.*
* *The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.*
* *If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.*
* *To compile the Java programs with package statements, you have to use -d option as shown below.*

*javac -d Destination\_folder file\_name.java*

* *Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.*

### *Example*

* *Let us look at an example that creates a package called animals. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.*

*Following package example contains interface named animals −*

*/\* File name : Animal.java \*/*

*package animals;*

*interface Animal {*

*public void eat();*

*public void travel();*

*}*

*Now, let us implement the above interface in the same package animals −*

*package animals;*

*/\* File name : MammalInt.java \*/*

*public class MammalInt implements Animal {*

*public void eat() {*

*System.out.println("Mammal eats");*

*}*

*public void travel() {*

*System.out.println("Mammal travels");*

*}*

*public int noOfLegs() {*

*return 0;*

*}*

*public static void main(String args[]) {*

*MammalInt m = new MammalInt();*

*m.eat();*

*m.travel();*

*}*

*}*

*Now compile the java files as shown below −*

*$ javac -d . Animal.java*

*$ javac -d . MammalInt.java*

*Now a package/folder with the name animals will be created in the current directory and these class files will be placed in it as shown below.*

*You can execute the class file within the package and get the result as shown below.*

*Mammal eats*

*Mammal travels*

## *9 The import Keyword*

* *If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.*

### *Example*

* *Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.*

*package payroll;*

*public class Boss {*

*public void payEmployee(Employee e) {*

*e.mailCheck();*

*}*

*}*

* *What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.*
* ***The fully qualified name of the class can be used.*** *For example −*

*payroll.Employee*

* ***The package can be imported using the import keyword and the wild card (\*)****. For example −*

*import payroll.\*;*

* ***The class itself can be imported using the import keyword.*** *For example −*

*import payroll.Employee;*

***Note − A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.***

## *Simple example of java package*

*The package keyword is used to create a package in java.*

*//save as Simple.java*

*package mypack;*

*public class Simple{*

*public static void main(String args[]){*

*System.out.println("Welcome to package");*

*}*

*}*

## *How to compile java package*

*If you are not using any IDE, you need to follow the syntax given below:*

*javac -d directory javafilename*

*For example*

*javac -d . Simple.java*

*The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).*

## *How to run java package program*

*You need to use fully qualified name e.g. mypack.Simple etc to run the class.*

|  |
| --- |
| ***To Compile:****javac -d . Simple.java* |
| ***To Run:****java mypack.Simple* |

*Output:Welcome to package*

|  |
| --- |
| *The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.* |

***10 How to access package from another package?***

*There are three ways to access the package from outside the package.*

*import package.\*;*

*import package.classname;*

*fully qualified name.*

***1) Using packagename.\****

* *If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.*
* *The import keyword is used to make the classes and interface of another package accessible to the current package.*

*Example of package that import the packagename.\**

*//save by A.java*

*package pack;*

*public class A{*

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

*}*

*//save by B.java*

*package mypack;*

*import pack.\*;*

*class B{*

*public static void main(String args[]){*

*A obj = new A();*

*obj.msg();*

*}*

*}*

*Output:Hello*

*2) Using packagename.classname*

* *If you import package.classname then only declared class of this package will be accessible.*

*Example of package by import package.classname*

*//save by A.java*

*package pack;*

*public class A{*

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

*}*

*//save by B.java*

*package mypack;*

*import pack.A;*

*class B{*

*public static void main(String args[]){*

*A obj = new A();*

*obj.msg();*

*}*

*}*

*Output:Hello*

***3) Using fully qualified name***

* *If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.*
* *It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.*

*Example of package by import fully qualified name*

*//save by A.java*

*package pack;*

*public class A{*

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

*}*

*//save by B.java*

*package mypack;*

*class B{*

*public static void main(String args[]){*

*pack.A obj = new pack.A();//using fully qualified name*

*obj.msg();*

*}*

*}*

*Output:Hello*

***Note: If you import a package, subpackages will not be imported.***

* *If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.*

***Note: Sequence of the program must be package then import then class.***

***11 sequence of package***

*Subpackage in java*

*Package inside the package is called the subpackage. It should be created to categorize the package further.*

*Example of Subpackage*

*package outer.inner;*

*class Simple{*

*public static void main(String args[]){*

*System.out.println("Hello subpackage");*

*}*

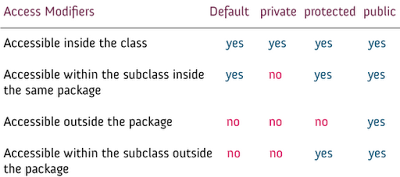
*}*

*To Compile: javac -d . Simple.java*

*To Run: java outer.inner.Simple*

*Output:Hello subpackage*

***12 Access Specifiers In Java***

* *Java Access Specifiers (also known as Visibility Specifiers ) regulate access to classes, fields and methods in Java.These Specifiers determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class.*
* *Access Specifiers can be used to restrict access. Access Specifiers are an integral part of object-oriented programming.  
    
  Types Of Access Specifiers :  
    
  In java we have four Access Specifiers and they are listed below.  
  1. public  
  2. private  
  3. protected  
  4. default(no specifier)  
    
  We look at these Access Specifiers in more detail.*[](http://2.bp.blogspot.com/-LTO8bwD3c6o/Tw1GoeAlHeI/AAAAAAAABlE/0EX24ENt9uY/s1600/access+specifiers.png)***public specifiers :***
* *Public Specifiers achieves the highest level of accessibility. Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.  
    
  Example*

*public class Demo { // public class  
public x, y, size; // public instance variables  
}*

***private specifiers :***

* *Private Specifiers achieves the lowest level of accessibility.private methods and fields can only be accessed within the same class to which the methods and fields belong.*
* *private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier.*
* *Using Private Specifier we can achieve encapsulation and hide data from the outside world.  
    
  Example :*

*public class Demo { // public class  
private double x, y; // private (encapsulated) instance variables  
  
public set(int x, int y) { // setting values of private fields  
this.x = x;  
this.y = y;  
}  
  
public get() { // setting values of private fields  
return Point(x, y);  
}  
}*

***protected specifiers :***

* *Methods and fields declared as protected can only be accessed by the subclasses in other package or any class within the package of the protected members' class. The protected access specifier cannot be applied to class and interfaces.*

*public class Demo {*

*protected x, y, size;*

*}*

***default(no specifier):***

* *When you don't set access specifier for the element, it will follow the default accessibility level. There is no default specifier keyword. Classes, variables, and methods can be default accessed.*
* *Using default specifier we can access class, method, or field which belongs to same package,but not from outside this package.  
    
  Example :*

*class Demo  
{  
int i; (Default)  
}*

# *13 Java - Interfaces*

* *An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.*
* *Along with abstract methods, an interface may also contain constants and nested types.*
* *Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.*
* *Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.*

*An interface is similar to a class in the following ways −*

* *An interface can contain any number of methods.*
* *An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.*
* *The byte code of an interface appears in a .class file.*
* *Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.*

*However, an interface is different from a class in several ways, including −*

* *You cannot instantiate an interface.*
* *An interface does not contain any constructors.*
* *All of the methods in an interface are abstract.*
* *An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.*
* *An interface is not extended by a class; it is implemented by a class.*
* *An interface can extend multiple interfaces.*

## *Declaring Interfaces*

* *The interface keyword is used to declare an interface. Here is a simple example to declare an interface −*

### *Example*

*Following is an example of an interface −*

*/\* File name : NameOfInterface.java \*/*

*import java.lang.\*;*

*// Any number of import statements*

*public interface NameOfInterface {*

*// Any number of final, static fields*

*// Any number of abstract method declarations\*

*}*

***Interfaces have the following properties −***

* *An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.*
* *Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.*
* *Methods in an interface are implicitly public.*

### *Example*

*/\* File name : Animal.java \*/*

*interface Animal {*

*public void eat();*

*public void travel();*

*}*

## *Implementing Interfaces*

* *When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.*
* *A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.*

### *Example*

*/\* File name : MammalInt.java \*/*

*public class MammalInt implements Animal {*

*public void eat() {*

*System.out.println("Mammal eats");*

*}*

*public void travel() {*

*System.out.println("Mammal travels");*

*}*

*public int noOfLegs() {*

*return 0;*

*}*

*public static void main(String args[]) {*

*MammalInt m = new MammalInt();*

*m.eat();*

*m.travel();*

*}*

*}*

*This will produce the following result −*

### *Output*

*Mammal eats*

*Mammal travels*

*When overriding methods defined in interfaces, there are several rules to be followed −*

* *The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.*
* *An implementation class itself can be abstract and if so, interface methods need not be implemented.*

*When implementation interfaces, there are several rules −*

* *A class can implement more than one interface at a time.*
* *A class can extend only one class, but implement many interfaces.*
* *An interface can extend another interface, in a similar way as a class can extend another class.*

***Interface Instances***

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

*Animal A = new MammalInt();*

*A.eat();*

*A.travel();*

* *Notice how the variable is declared to be of the interface type Animal while the object created is of type MammalInt. Java allows this because the class MammalInt implements the MyInterface interface.*
* *You can then reference instances of the MammalInt class as instances of the Animal  interface.*
* *You cannot create instances of a Java interface by itself. You must always create an instance of some class that implements the interface, and reference that instance as an instance of the interface*

## *14 Extending Interfaces*

* *An interface can extend another interface in the same way that a class can extend another class. The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.*

*The following Sports interface is extended by Hockey and Football interfaces.*

### *Example*

*// Filename: Sports.java*

*public interface Sports {*

*public void setHomeTeam(String name);*

*public void setVisitingTeam(String name);*

*}*

*// Filename: Football.java*

*public interface Football extends Sports {*

*public void homeTeamScored(int points);*

*public void visitingTeamScored(int points);*

*public void endOfQuarter(int quarter);*

*}*

*// Filename: Hockey.java*

*public interface Hockey extends Sports {*

*public void homeGoalScored();*

*public void visitingGoalScored();*

*public void endOfPeriod(int period);*

*public void overtimePeriod(int ot);*

*}*

* *The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.*

## *Extending Multiple Interfaces*

* *A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.*
* *The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.*
* *For example, if the Hockey interface extended both Sports and Event, it would be declared as −*

### *Example*

*public interface Hockey extends Sports, Event*

# *15 Nested or Inner interfaces in Java*

* *An interface which is declared inside another interface or class is called nested interface. They are also known as inner*[*interface*](http://beginnersbook.com/2013/05/java-interface/)*. Since nested interface cannot be accessed directly, the main purpose of using them is to resolve the namespace by grouping related interfaces (or related interface and class) together.*
* *This way, we can only call the nested interface by using outer class or outer interface name followed by dot( . ), followed by the interface name.*

*Example: Entry interface inside Map interface is nested. Thus we access it by calling [Map.Entry](http://beginnersbook.com/2014/06/map-entry-interface-in-java/" \t "_blank).*

*Note:*

* *Nested interfaces are static by default. You don’t have to mark them static explicitly as it would be redundant.*
* *Nested interfaces declared inside class can take any access modifier, however nested interface declared inside interface is public implicitly.*

### *Example 1: Nested interface declared inside another interface*

*interface MyInterfaceA{*

*void display();*

*interface MyInterfaceB{*

*void myMethod();*

*}*

*}*

*class NestedInterfaceDemo1  implements MyInterfaceA.MyInterfaceB*

*{*

*public void myMethod(){*

*System.out.println("Nested interface method");*

*}*

*public static void main(String args[]){*

*MyInterfaceA.MyInterfaceB obj=   new NestedInterfaceDemo1();*

*obj.myMethod();*

*}*

*}*

*Output:  
Nested interface method*

### *Example 2: Nested interface declared inside a class*

*class MyClass{*

*interface MyInterfaceB{*

*void myMethod();*

*}*

*}*

*class NestedInterfaceDemo2 implements MyClass.MyInterfaceB{*

*public void myMethod(){*

*System.out.println("Nested interface method");*

*}*

*public static void main(String args[]){*

*MyClass.MyInterfaceB obj=  new NestedInterfaceDemo2();*

*obj.myMethod();*

*}*

*}*

*Output:  
Nested interface method*

# *16* [*Accessing Interface Variables in Java*](http://www.xcnotes.com/interfaces-in-java/accessing-interface-variables)

* *The variables of an interface are always declared as"final". Final variables are those variables, whose values are constants and cannot be changed.*
* *The class that implements the interface can use the variables as declared in the interface and cannot modify or changed the value of the variable.*

*interface selectcolor  
{  
int blue=4;  
int yellow=5;  
int pink=6;  
public void choose(int color);  
}  
class selectimp implements selectcolor  
{  
public void choose(int color)  
{  
switch(color)  
{  
case blue:  
   System.out.println("The color selected is blue");  
   break;  
case yellow:  
   System.out.println("The color selected is yellow");  
   break;  
case pink:  
   System.out.println("The color selected is pink");  
   break;  
}  
}  
public static void main(String aa[])  
{  
  
selectimp st=new selectimp();  
st.choose(5);  
st.choose(4);  
st.choose(6);  
}  
}*

*The color selected is yellow*

*The color selected is blue*

*The color selected is pink*

## *Explanation....*

* *In this example first  an interface selectcolor is created and the value for the integers yellow,blue, pink are set as 4,5,6. Then a method choose() which takes an integer parameter is declared.*
* *A class selectimp is created, which implements the interface selectcolor.  
  Then the method choose() of the interface selectcolor is implemented using the switch case statements.*
* *Then there is a main() which creates the object of the class selectimp and call the choose() of the selectimp class with differentparameters or arguments.*

# *17 Java - Exceptions*

* *An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.*
* *An exception can occur for many different reasons. Following are some scenarios where an exception occurs.*
* *A user has entered an invalid data.*
* *A file that needs to be opened cannot be found.*
* *A network connection has been lost in the middle of communications or the JVM has run out of memory.*
* *Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner*
* *Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.*
* ***Checked exceptions****− A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.*
* *For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.*

### *Example*

*import java.io.File;*

*import java.io.FileReader;*

*public class FilenotFound\_Demo {*

*public static void main(String args[]) {*

*File file = new File("E://file.txt");*

*FileReader fr = new FileReader(file);*

*}*

*}*

*If you try to compile the above program, you will get the following exceptions.*

### *Output*

*C:\>javac FilenotFound\_Demo.java*

*FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown*

*FileReader fr = new FileReader(file);*

*^*

*1 error*

*Note − Since the methods read() and close() of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.*

* ***Unchecked exceptions****− An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.*
* *For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsException exception occurs.*

### *Example*

*public class Unchecked\_Demo {*

*public static void main(String args[]) {*

*int num[] = {1, 2, 3, 4};*

*System.out.println(num[5]);*

*}*

*}*

*If you compile and execute the above program, you will get the following exception.*

### *Output*

*Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5*

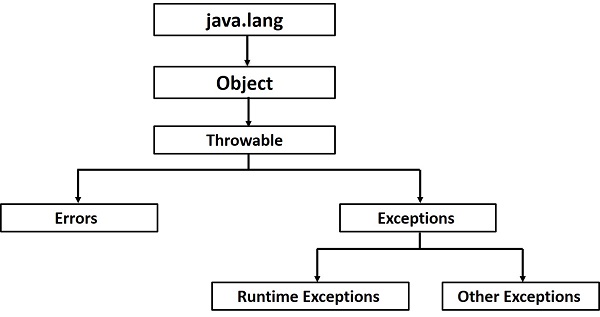
*at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)*

* ***Errors****− These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.*

## *Exception Hierarchy*

* *All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.*
* *Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.*

*The Exception class has two main subclasses: IOException class and RuntimeException Class.*

**

*Following is a list of most common checked and unchecked*[*Java's Built-in Exceptions*](https://www.tutorialspoint.com/java/java_builtin_exceptions.htm)*.*

|  |
| --- |
| ***Exception & Description*** |
| *1* | ***ArithmeticException***  *Arithmetic error, such as divide-by-zero.* |
| *2* | ***ArrayIndexOutOfBoundsException***  *Array index is out-of-bounds.* |
| *3* | ***ArrayStoreException***  *Assignment to an array element of an incompatible type.* |
| *4* | ***ClassCastException***  *Invalid cast.* |
| *5* | ***IllegalArgumentException***  *Illegal argument used to invoke a method.* |
| *6* | ***IllegalMonitorStateException***  *Illegal monitor operation, such as waiting on an unlocked thread.* |
| *7* | ***IllegalStateException***  *Environment or application is in incorrect state.* |
| *8* | ***IllegalThreadStateException***  *Requested operation not compatible with the current thread state.* |
| *9* | ***IndexOutOfBoundsException***  *Some type of index is out-of-bounds.* |
| *10* | ***NegativeArraySizeException***  *Array created with a negative size.* |
| *11* | ***NullPointerException***  *Invalid use of a null reference.* |
| *12* | ***NumberFormatException***  *Invalid conversion of a string to a numeric format.* |
| *13* | ***SecurityException***  *Attempt to violate security.* |
| *14* | ***StringIndexOutOfBounds***  *Attempt to index outside the bounds of a string.* |
| *15* | ***UnsupportedOperationException***  *An unsupported operation was encountered.* |

*Following is the list of Java Checked Exceptions Defined in java.lang.*

|  |  |
| --- | --- |
| ***Sr.No.*** | ***Exception & Description*** |
| *1* | ***ClassNotFoundException***  *Class not found.* |
| *2* | ***CloneNotSupportedException***  *Attempt to clone an object that does not implement the Cloneable interface.* |
| *3* | ***IllegalAccessException***  *Access to a class is denied.* |
| *4* | ***InstantiationException***  *Attempt to create an object of an abstract class or interface.* |
| *5* | ***InterruptedException***  *One thread has been interrupted by another thread.* |
| *6* | ***NoSuchFieldException***  *A requested field does not exist.* |
| *7* | ***NoSuchMethodException***  *A requested method does not exist.* |

## *Exceptions Methods*

*Following is the list of important methods available in the Throwable class.*

|  |  |
| --- | --- |
| *Sr.No.* | *Method & Description* |
| *1* | *public String getMessage()*  *Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.* |
| *2* | *public Throwable getCause()*  *Returns the cause of the exception as represented by a Throwable object.* |
| *3* | *public String toString()*  *Returns the name of the class concatenated with the result of getMessage().* |
| *4* | *public void printStackTrace()*  *Prints the result of toString() along with the stack trace to System.err, the error output stream.* |
| *5* | *public StackTraceElement [] getStackTrace()*  *Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.* |
| *6* | *public Throwable fillInStackTrace()*  *Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.* |

## *18 Exception Handling IN JAVA*

## *Exceptions can be handled by using 'try-catch' block. Try block contains the code which is under observation for exceptions. The catch block contains the remedy for the exception. If any exception occurs in the try block then the control jumps to catch block.*

## *If a method doesn't handle the exception, then it is mandatory to specify the exception type in the method signature using 'throws' clause.*

## *We can explicitly throw an exception using 'throw' clause*

## *Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.*

## 

## *try/catch*

* *A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception.*
* *Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −*

### *Syntax*

*try {*

*// Protected code*

*}catch(ExceptionName e1) {*

*// Catch block*

*}*

* *The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.*
* *A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.*

### *Example*

*The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.*

*// File Name : ExcepTest.java*

*import java.io.\*;*

*public class ExcepTest {*

*public static void main(String args[]) {*

*try {*

*int a[] = new int[2];*

*System.out.println("Access element three :" + a[3]);*

*}catch(ArrayIndexOutOfBoundsException e) {*

*System.out.println("Exception thrown :" + e);*

*}*

*System.out.println("Out of the block");*

*}*

*}*

*This will produce the following result −*

### *Output*

*Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3*

*Out of the block*

## *Multiple Catch Blocks*

*A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −*

### *Syntax*

*try {*

*// Protected code*

*}catch(ExceptionType1 e1) {*

*// Catch block*

*}catch(ExceptionType2 e2) {*

*// Catch block*

*}catch(ExceptionType3 e3) {*

*// Catch block*

*}*

* *The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list.*
* *If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.*

### *Example*

*Here is code segment showing how to use multiple try/catch statements.*

*try {*

*file = new FileInputStream(fileName);*

*x = (byte) file.read();*

*}catch(IOException i) {*

*i.printStackTrace();*

*return -1;*

*}catch(FileNotFoundException f) // Not valid! {*

*f.printStackTrace();*

*return -1;*

*}*

## *The Throws/Throw Keywords*

* *If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.*
* *You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.*
* *Try to understand the difference between throws and throw keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.*

*The following method declares that it throws a RemoteException −*

### *Example*

*import java.io.\*;*

*public class className {*

*public void deposit(double amount) throws RemoteException {*

*// Method implementation*

*throw new RemoteException();*

*}*

*// Remainder of class definition*

*}*

* *A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException −*

### *Example*

*import java.io.\*;*

*public class className {*

*public void withdraw(double amount) throws RemoteException, InsufficientFundsException*

*{*

*// Method implementation*

*}*

*// Remainder of class definition*

*}*

## *The Finally Block*

* *The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.*
* *A finally block appears at the end of the catch blocks and has the following syntax −*

### *Syntax*

*try {*

*// Protected code*

*}catch(ExceptionType1 e1) {*

*// Catch block*

*}catch(ExceptionType2 e2) {*

*// Catch block*

*}catch(ExceptionType3 e3) {*

*// Catch block*

*}finally {*

*// The finally block always executes.*

*}*

### *Example*

*public class ExcepTest {*

*public static void main(String args[]) {*

*int a[] = new int[2];*

*try {*

*System.out.println("Access element three :" + a[3]);*

*}catch(ArrayIndexOutOfBoundsException e) {*

*System.out.println("Exception thrown :" + e);*

*}finally {*

*a[0] = 6;*

*System.out.println("First element value: " + a[0]);*

*System.out.println("The finally statement is executed");*

*}*

*}*

*}*

*This will produce the following result −*

### *Output*

*Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3*

*First element value: 6*

*The finally statement is executed*

*Note the following −*

* *A catch clause cannot exist without a try statement.*
* *It is not compulsory to have finally clauses whenever a try/catch block is present.*
* *The try block cannot be present without either catch clause or finally clause.*
* *Any code cannot be present in between the try, catch, finally blocks.*

## *19 User-defined Exceptions*

* *You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −*
* *All exceptions must be a child of Throwable.*
* *If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.*
* *If you want to write a runtime exception, you need to extend the RuntimeException class.*

*We can define our own Exception class as below −*

*class MyException extends Exception {*

*}*

* *You just need to extend the predefined Exception class to create your own Exception. These are considered to be checked exceptions.*
* *The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.*

### *Example*

*// File Name InsufficientFundsException.java*

*import java.io.\*;*

*public class InsufficientFundsException extends Exception {*

*private double amount;*

*public InsufficientFundsException(double amount) {*

*this.amount = amount;*

*}*

*public double getAmount() {*

*return amount;*

*}*

*}*

* *To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.*

*// File Name CheckingAccount.java*

*import java.io.\*;*

*public class CheckingAccount {*

*private double balance;*

*private int number;*

*public CheckingAccount(int number) {*

*this.number = number;*

*}*

*public void deposit(double amount) {*

*balance += amount;*

*}*

*public void withdraw(double amount) throws InsufficientFundsException {*

*if(amount <= balance) {*

*balance -= amount;*

*}else {*

*double needs = amount - balance;*

*throw new InsufficientFundsException(needs);*

*}*

*}*

*public double getBalance() {*

*return balance;*

*}*

*public int getNumber() {*

*return number;*

*}*

*}*

*The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.*

*// File Name BankDemo.java*

*public class BankDemo {*

*public static void main(String [] args) {*

*CheckingAccount c = new CheckingAccount(101);*

*System.out.println("Depositing $500...");*

*c.deposit(500.00);*

*try {*

*System.out.println("\nWithdrawing $100...");*

*c.withdraw(100.00);*

*System.out.println("\nWithdrawing $600...");*

*c.withdraw(600.00);*

*}catch(InsufficientFundsException e) {*

*System.out.println("Sorry, but you are short $" + e.getAmount());*

*e.printStackTrace();*

*}*

*}*

*}*

*Compile all the above three files and run BankDemo. This will produce the following result −*

### *Output*

*Depositing $500...*

*Withdrawing $100...*

*Withdrawing $600...*

*Sorry, but you are short $200.0*

*InsufficientFundsException*

*at CheckingAccount.withdraw(CheckingAccount.java:25)*

*at BankDemo.main(BankDemo.java:13)*

***20 Chained Exceptions in Java***

* *Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception.*
* *For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.*
* *The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.*

***Constructors Of Throwable class Which support chained exceptions in java*** *:*

1. ***Throwable(Throwable cause)*** *:- Where cause is the exception that causes the current exception.*
2. ***Throwable(String msg, Throwable cause)*** *:- Where msg is the exception message and cause is the exception that causes the current exception.*

*Methods Of Throwable class Which support chained exceptions in java :*

1. ***getCause()*** *method :- This method returns actual cause of an exception.*
2. ***initCause(Throwable cause)*** *method :- This method sets the cause for the calling exception.*

***Example of using Chained Exception****:*

|  |
| --- |
| *// Java program to demonstrate working of chained exceptions*  *public class ExceptionHandling*  *{*  *public static void main(String[] args)*  *{*  *try*  *{*  *// Creating an exception*  *NumberFormatException ex =*  *new NumberFormatException("Exception");*    *// Setting a cause of the exception*  *ex.initCause(new NullPointerException(*  *"This is actual cause of the exception"));*    *// Throwing an exception with cause.*  *throw ex;*  *}*    *catch(NumberFormatException ex)*  *{*  *// displaying the exception*  *System.out.println(ex);*    *// Getting the actual cause of the exception*  *System.out.println(ex.getCause());*  *}*  *}*  *}* |

*Output:*

*java.lang.NumberFormatException: Exception*

*java.lang.NullPointerException: This is actual cause of the exception*

# *21 Object class in Java*

* ***Object****class is present in****java.lang****package. Every class in Java is directly or indirectly derived from the****Object****class.*
* *If a Class does not extend any other class then it is direct child class of****Object****and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.*

***Using Object class methods***

*There are several methods in****Object****class:*

* ***toString()****: toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance*
* ***hashCode()****: For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm.*
* ***equals(Object obj)****: Compares the given object to “this” object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override****equals(Object obj)****method to get our own equality condition on Objects.*
* ***getClass()****: Returns the class object of “this” object and used to get actual runtime class of the object. It can also be used to get metadata of this class.*
* ***finalize()****method : This method is called just before an object is garbage collected. It is called by the*[*Garbage Collector*](http://www.geeksforgeeks.org/garbage-collection-java/)*on an object when garbage collector determines that there are no more references to the object*
* ***clone()****: It returns a new object that is exactly the same as this object. For clone() method refer*[*Clone()*](http://www.geeksforgeeks.org/clone-method-in-java-2/)
* *The remaining three methods****wait()****,****notify()******notifyAll()****are related to Concurrency.*

# *22 java.lang.Object.finalize() Method*

* *The java.lang.Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.*

## *Declaration*

*Following is the declaration for java.lang.Object.finalize() method*

*protected void finalize()*

## *Example*

*The following example shows the usage of lang.Object.finalize() method.*

*import java.util.\*;*

*public class ObjectDemo extends GregorianCalendar {*

*public static void main(String[] args) {*

*try {*

*// create a new ObjectDemo object*

*ObjectDemo cal = new ObjectDemo();*

*// print current time*

*System.out.println("" + cal.getTime());*

*// finalize cal*

***System.out.println("Finalizing...");***

***cal.finalize();***

***System.out.println("Finalized.");***

*} catch (Throwable ex) {*

*ex.printStackTrace();*

*}*

*}*

*}*

*Let us compile and run the above program, this will produce the following result −*

*Sat april 22 00:27:21 EEST 2017*

*Finalizing...*

*Finalized.*