RAGHU

ENGINEERING COLLEGE

(AUTONOMOUS | VISAKHAPATNAM)

Unit-III
Inheritance and Exception Handling

Inheritance

• Inheritance is a concept where new classes can be produced from existing classes. The newly created class acquires all the features of an existing class.

• A class that is inherited is called as ‘super class’ or ‘base class’ or ‘parent class’. The class that does the inheriting is called as ‘sub class’ or ‘derived class’ or ‘child class’.

• Inheritance allows sub classes to inherit all the variables and methods of their parent class.

Inheritance

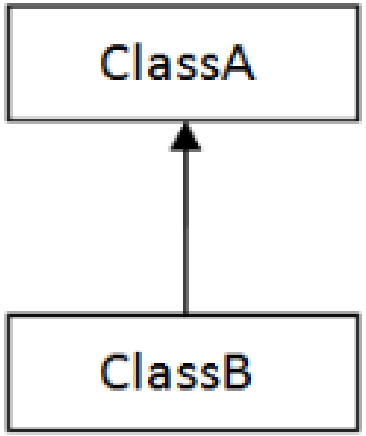
A sub class can be defined by the use of a keyword ‘extends’ as below:

```
class sub_class_name extends super_class_name
{
        Variable declaration;
        Method declaration;
}
```

The keyword ‘extends’ signifies that the properties of the super class are extended to the sub class. The sub class will now contain its own variables and methods as well as those of the super class. But it is not vice-versa.

Single or Simple Inheritance

Deriving a sub class from only one super class is called 'Single or Simple Inheritance'.



Refer Program

Hierarchical Inheritance

Deriving two or more number of classes from a single super class is called ‘Hierarchical Inheritance’.

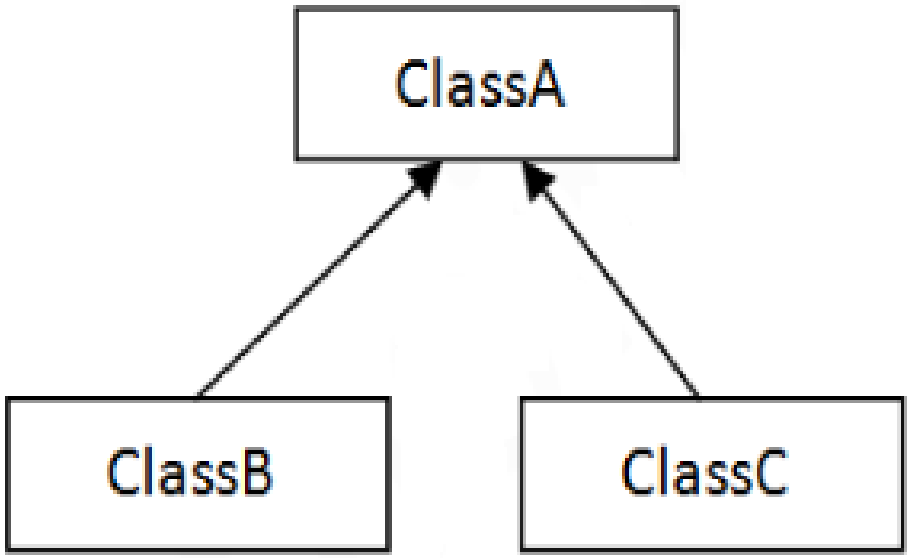


Fig. Hierarchical Inheritance

Refer Program

Multiple Inheritance

Derivation of one class from two or more super classes is called 'Multiple Inheritance'. But Java does not support multiple inheritance. Syntactically we can achieve multiple inheritance by using interfaces but the actual implementation of multiple inheritance is not there.

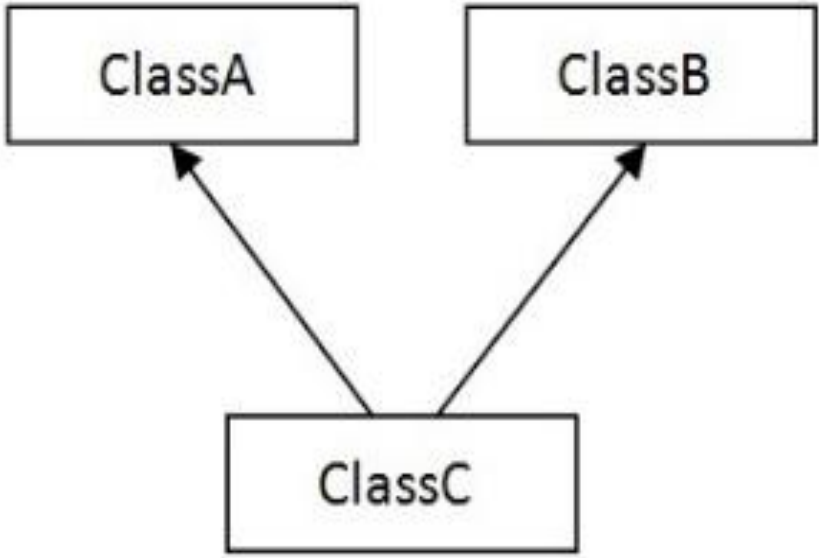


Fig. Multiple Inheritance

Multilevel Inheritance

Derivation of a class from another derived class is called 'Multilevel Inheritance'. Derived class is a class which is derived from another class.

Refer Program

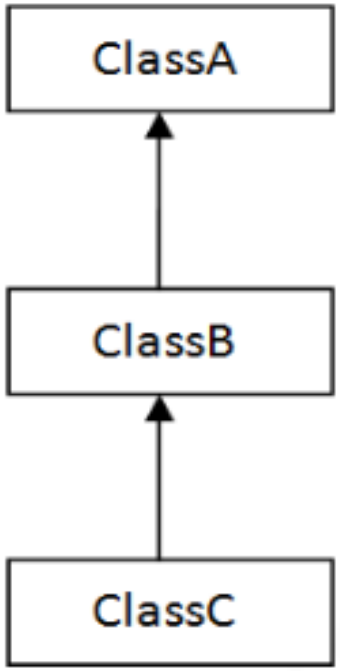


Fig. Multilevel Inheritance

Hybrid Inheritance

Derivation of a class involving more than one form of Inheritance is called ‘Hybrid Inheritance’.

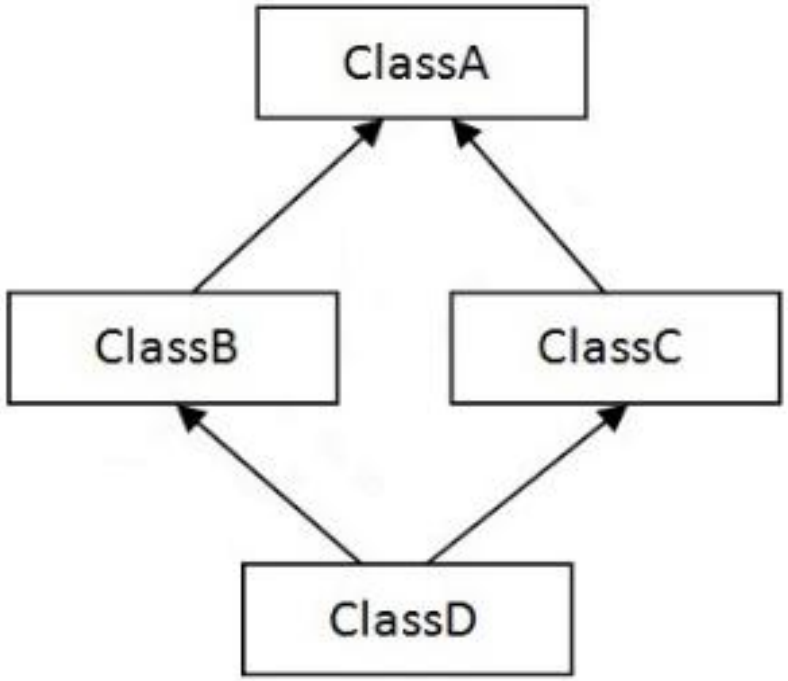


Fig. Hybrid Inheritance

Polymorphism

- The ability to exist in different forms is called 'Polymorphism'.
- In Java, a variable, an object or a method can exist in different forms, thus performing various tasks depending on the context.
- In Polymorphism there are two types, they are
    1. Static Polymorphism
    2. Dynamic Polymorphism

Polymorphism

Static Polymorphism:

The polymorphism exhibited at the compile time is called as 'Static or Compile Time Polymorphism'. Method overloading is the example for Static polymorphism.

Dynamic Polymorphism:

The polymorphism exhibited at the run time is called as 'Dynamic or Run Time Polymorphism'. Method overriding is the example for Dynamic polymorphism.

Dynamic method dispatch:

It is the mechanism by which a call to an **overridden method** is resolved at run time, rather than compile time.

Method Overloading

The process in which writing two or more methods in the same class in such a way that each method has same name but with different method signature is called 'Method Overloading'.

Refer Program

Method Overriding

The process in which writing two or more methods in super and sub classes such that the methods have same name and same signature, is called ‘Method Overriding’.

Refer Program

super keyword

The ‘super’ is a keyword which can be used to refer the object of immediate super class.

If we create an object to the super class, we can access only the super class members. But, if we create sub class object, all the members of both super and sub class are available to it. That is why we always create an object to the sub class in inheritance.

Sometimes, the super class members and sub class members may have same names. In that case, by default super class members are overridden and only the sub class members are accessible.

super keyword

It can be used with variables, methods and to invoke super class constructor.

To access super class variable:

super.variable;

To access super class method:

super.method_name;

To invoke super class constructor:

super(); //refers to super class default constructor

super(value);//refers to super class parameterized constructor

super keyword

program to access super class variables and methods

program to invoke super class default constructor

program to invoke super class parameterized constructor

final keyword

The 'final' is a non-access modifier keyword. It is used for finalizing the implementations of classes, methods, and variables. which is used with methods to prevent overriding, with classes to prevent inheritance and with variables to restrict the changing of a variable value.

Usage of 'final' with variables:

If a variable is declared as 'final' then it cannot be modified throughout the program further.

Syntax:

final datatype variable_name=value;

Example:

final float PI=3.1416;

final keyword

Usage of 'final' with methods:

If a method is declared as the 'final' then it cannot be overridden.

Syntax:

```
final return_type method_name(parameters)
{
        //body
}
```

final keyword

Usage of 'final' with classes:

If a class is defined using 'final' keyword then it will not allow other classes to inherit it. This mean that sub classes cannot be created to a final class.

Syntax:

final class class_name

Refer Program

Abstract class

Abstract method:

An abstract method is a method without method body or implementation. An abstract method is written when the same method has to perform different tasks depending on the object calling it.

An abstract method doesn't contain any body. It contains only the method header. So, we can say it is an incomplete method. Abstract methods declared by using a keyword 'abstract'.

Syntax:

abstract return_type method_name();

Example:

abstract void calculate();

Abstract class

An abstract class is a class that contains zero or more abstract methods. It means that an abstract class generally contains some abstract methods. These abstract methods need to be implemented in the sub class.

Syntax:

```
abstract class class_name
{
        abstract return_type method_name();
}
```

Refer Program

Abstract class

We cannot create the object for abstract classes but we can create reference to the abstract class. That reference can be used to call the method of some other class.

Example snippet:

```
BaseClass b=new SubClass();
b.calculate(4);
```

In the above snippet, ‘b’ is the reference of the abstract class BaseClass but stores the object of SubClass.

Interfaces

An interface contains only abstract methods which are all incomplete methods. So, it is not possible to create an object for an interface.

In this case we can create separate classes where we can implement all the methods of an interface. Those classes are called ‘Implementation Classes’. Since, implementation classes will have all the methods with body, and it is possible to create objects to the implementation classes.

Methods inside the interface are public and abstract since they should be available to third party vendors to provide the implementation.

Interfaces

Syntax:

```
interface interface_name
{
        return_type method_name();
}
```

Implementing an Interface:

Once an interface is defined, one or more classes can implement that interface. A class can implement an interface by using a keyword 'implements'.

Syntax:

```
class class_name implements interface_name1, interface_name2….
{
        //implementation
}
```

Interfaces

The methods that implement an interface must be declared as public and also type signature of the implementing method must match exactly with the type signature specified in the interface definition.

Note: By default the variables of the interface are final.

Refer Program

Multiple Inheritance using Interfaces

In multiple inheritance sub classes are derived from multiple super classes. If two super classes have same name for their members (variables and methods) then which member to be inherited into the sub class is the main confusion in multiple inheritance.

The syntactical implementation of the multiple inheritance is achieved by interfaces. When implementing a sub class from more than one super class this confusion is resolved by using dot operator to access the members as below:

interface_name.member_name;

Refer Program

Abstract class vs Interface

- An abstract class written when there are some common features to be shared by all the objects.
- When abstract class is written, it is the duty of the programmer to provide the sub classes to it.
- An abstract class contain some abstract and some concrete methods

- An interface is written when all the features are to be implemented differently in the objects.
- An interface is written when the programmer wants to leave the implementation to the third party vendors.
- An interface contains only abstract methods.

Abstract class vs Interface

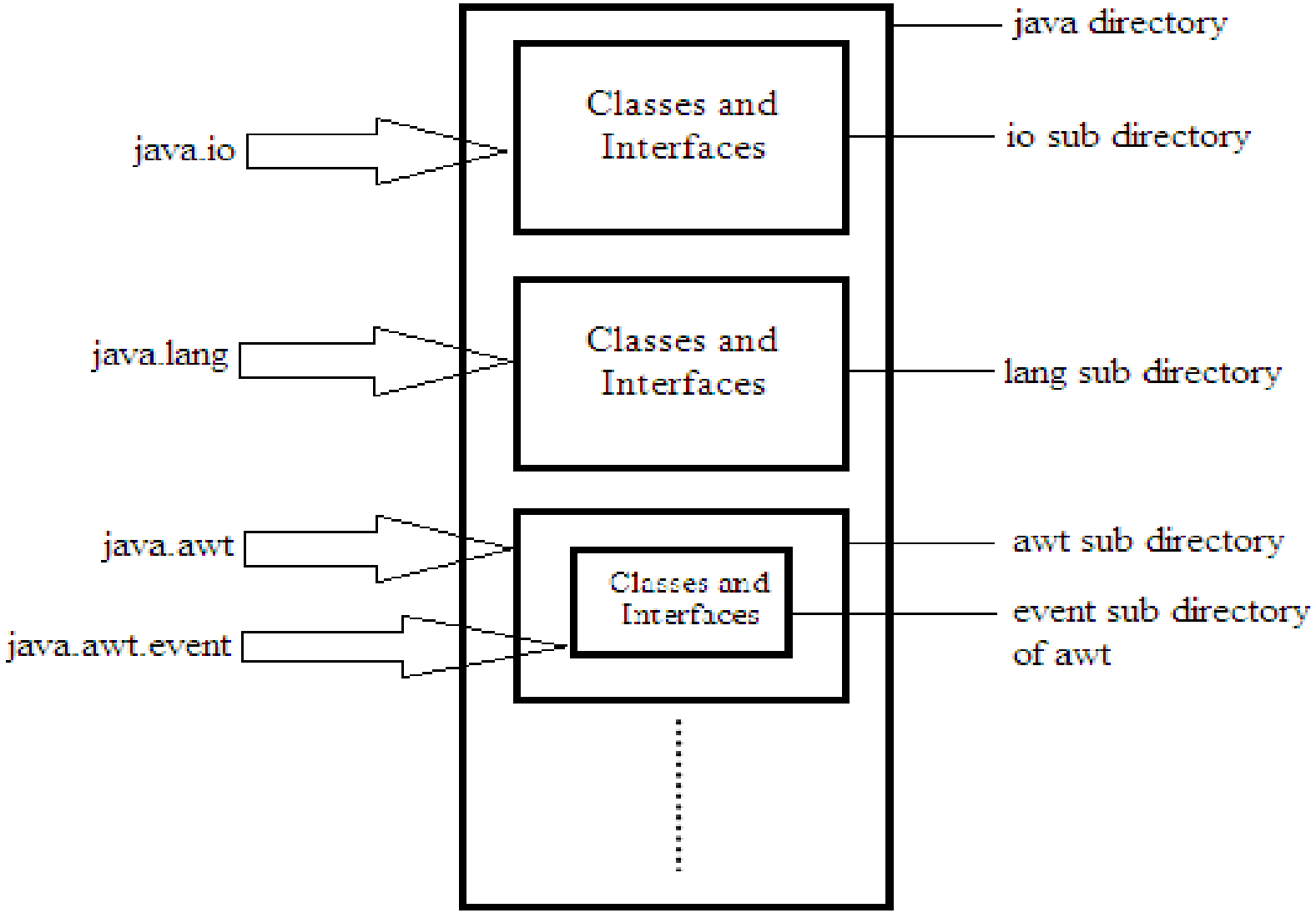
• An abstract class can contain instance variables.	• An interface contains final variables only.
• All the abstract methods of the abstract class should be implemented in its sub class.	• All the methods of an interface should be implemented in its implementation classes.
• Abstract class is declared by the use of keyword ‘abstract’.	• Interface is declared by the use of the keyword ‘interface’.

Packages

A package represents a directory to encapsulate a group of classes, interfaces and sub packages. For example when we write statement like, '**import java.io.*;**' we are importing the classes and interfaces of 'java.io.package' into our program.

In 'java.io.*', java is a directory name and 'io' is the sub directory in that. And '*' represents all the members of that 'io' subdirectory.

Packages


java directory
Classes and Interfaces
io sub directory
java.io
java.lang
Classes and Interfaces
lang sub directory
java.awt
Classes and Interfaces
awt sub directory
java.awt.event
event sub directory of awt

Advantages of Packages

- Packages are useful to put together all classes and interfaces in a package which are performing similar task.

- Packages hide the classes and interfaces in a separate sub directory so that accidental deletion of classes and interfaces will not take place.

- The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means we can use same names for classes and interfaces of two different packages.

- Packages can be created once and used several times.

Types of Packages

Types of packages:

There are two different types of packages available in Java. They are:

Built-in package

User-defined package

Built-in package:

Built-in packages are the packages which are already available in Java language. These packages provide almost all necessary classes, interfaces and methods to perform any task in the program.

Built in Packages

java.lang: lang stands for language. This package got primary classes and interfaces essential for developing a basic java program.

java.util: util stands for utility. This package contains classes like Stack, LinkedList, HashTable, Vectors, and Arrays etc. There are also classes for handling date and time operations.

java.io: io stands for input output. This package contains stream classes. A stream represents flow of data from one place to another place. Streams are useful to perform reading and writing related tasks.

Built in Packages

java.awt: awt stands for abstract window tool kit. This helps to develop graphical user interface of the java applications.

javax.swing: This is also useful to develop graphical user interface like java.awt. The ‘x’ in javax represents that is an extended package which got additional features.

java.net: net stands for network. Client-Server programming can be done by using this package.

Built in Packages

java.applet: Applets are also java programs which need to be loaded from the server into the client and get executed on the client machine on a network. Applet class of this package useful to create and use applets.

java.sql: sql stands for structured query language. This package helps to perform database related tasks.

Creating Packages

User-defined packages:

The users of java language can also create their own packages. They are called as 'User-defined packages'. User-defined packages can also be imported into another class and used exactly in the same ways as the built-in packages.

Creation of a package:

To create the package we need to use a keyword 'package' as below:

```
package package_name;  //to create a package
package package_name.sub_package_name; //to  create
                        //a package within a sub directory
```

Creating Packages

The package statement should be the first statement in the java program. While creating a class in a package all the members and the class are to be defined as ‘public’ except the instance variables.

Syntax to compile a package program:

The package program needs to be compiled as:

javac –d dest_directory_name filename.java

In the above statement, ‘-d’ represents creating a directory and dest_directory_name represents location where to store the newly generated class file and package.

creating Packages

```
package pack;
public class Addition
{
        public Addition()
        {
                System.out.println("This is Addition class constructor");
        }
        public void addMethod(int a,int b)
        {
                System.out.println("Addition is:"+(a+b));
        }
}
```

Output: D:\Java>javac -d D:/ Addition.java

This will create a package with name ‘pack’ and stores generated file ‘Addition.class’ in that package.

using Packages (importing)

First way: Create an object for the class as:

package_name.class_name object_name=new package_name.class_name();

Then access the methods of that class by the use object reference.

Example program:

```
class PackageDemo1
{
        public static void main(String args[])
        {
                pack.Addition obj=new pack.Addition();
                obj.addMethod(35,45);
        }
}
```

using Packages (importing)

Second way: Simply import the package and classes required as below,

import package_name.class_name;

Then access the methods of that class by the using object reference.

Example program:

```
import pack.Addition;
class PackageDemo1
{
	public static void main(String args[])
	{
		Addition obj=new Addition();
		obj.addMethod(35,45);
	}
}
```

Output:

```
This is Addition class constructor
Addition is:80
```

Adding new classes to package

```
package pack;
public class Subtraction
{
        public void subMethod(int a,int b)
        {
                System.out.println("Subtraction is:"+(a-b));
        }
}
```

If ‘pack’ is present then it will add the new class file to that existing package otherwise it will create a new package ‘pack’ and then it will add the new class file ‘Subtraction’ into that package.

Importance of CLASSPATH

CLASSPATH is an environmental variable that tells the Java compiler where to look for the class files and packages to import.

To set CLASSPATH in Windows Operating system:
ECHO %CLASSPATH% - It will return the current class path.
SET CLASSPATH=D:/;.;%CLASSPATH% - This command sets the class path to 'D:' directory and to the current working directory and also keep remains the already existing class path.
In Ubuntu Operating system:
export CLASSPATH=class_path:$CLASSPATH

Wrapper Class

- A Wrapper class is a class whose object wraps or contains a primitive data types.
- When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types.
- In other words, we can wrap a primitive value into a wrapper class object.
- There are three reasons that you might use Wrapper class object rather than a primitive:

1. As an argument of a method that expects an object.
2. To use constants defined by the class, such as *MIN_VALUE* and *MAX_VALUE*, that provide the upper and lower bounds of the data type.

Wrapper Class

3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

Few of the methods offer by these wrapper classes are given below,

static int parseInt(String s) - Returns an integer (decimal only).

static String toString(int i) - Returns a String object representing the specified integer.

Wrapper Class

- The eight classes of *java.lang* package are known as wrapper classes in java.
- The list of eight wrapper classes are given below:

Wrapper Class	Primitive Type
Boolean	boolean
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double

Wrapper Class

• **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as *Autoboxing*. For example – conversion of int to Integer, long to Long, double to Double etc.

Example snippet: char c1='A';
Character c2=c1;

• **Unboxing:** Automatically converting an object of a wrapper class to its corresponding primitive type is known as *Unboxing*. For example – conversion of Integer to int, Long to long, Double to double etc.

Example snippet: Character c3='P';
char c4=c3;

Java.lang package

The package 'java.lang' contains classes and interfaces that are fundamental to all of Java programming. This package is automatically imported into all Java programs. It is most widely used package.

Classes of java.lang package:

Boolean	Integer	Runtime	Thread
Byte	Math	RuntimePermmission	ThreadGroup
Character	Number	Short	ThreadLocal
Class	Long	StackTraceElement	Throwable
ClassLoader	Object	String	Void
Compiler	Package	StringBuffer	
Double	Process	StringBuilder	
Float	ProcessBuilder	System	

Java.lang package

Interfaces of java.lang package:

Appendable	Iterable
CharSequence	Readable
Cloneable	Runnable
Comparable	

Exception Handling

Exception:

An exception is an event that may cause abnormal termination of the program during its execution. Exceptions are categorized into two groups. They are:

Built-in Exceptions

User-defined Exceptions

Built-in Exceptions:

Built-in exceptions are the exceptions which are already available in Java. These exceptions are suitable to explain certain error situations. The following are the important Built-in exceptions.

Exception Handling

Exception classes:

ArithmeticException:

Raised when exception occurred in the arithmetic operation.

ArrayIndexOutOfBoundsException:

To indicate an array has been accessed with an illegal index.

ClassNotFoundException:

This is raised when we try to access a non existing class.

FileNotFoundException:

Raised when a file is not accessible or does not open.

Exception Handling

Exception classes:

IOException:

Raised when an input or output operation failed.

InterruptedException:

Raised when a thread is waiting, sleeping or it is interrupted.

NoSuchFieldException:

Raised when a class does not contain the field specified.

NoSuchMethodException:

Thrown when accessing a method that does not exist.

Exception Handling

Exception classes:

NullPointerException:

Raised when referring to the methods of a null object

NumberFormatException: Raised when a method could not convert a string value into numeric format.

StringIndexOutOfBoundsException:

Thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

Exception Handling

In exceptions there are two types. They are:

1. Checked Exceptions:

The exceptions which are raised at the compile time are called as 'Checked Exceptions' or 'Compile-time Exceptions'. If a checked exception is raised it should be handled or throw it out by the programmer.

Example: IOException

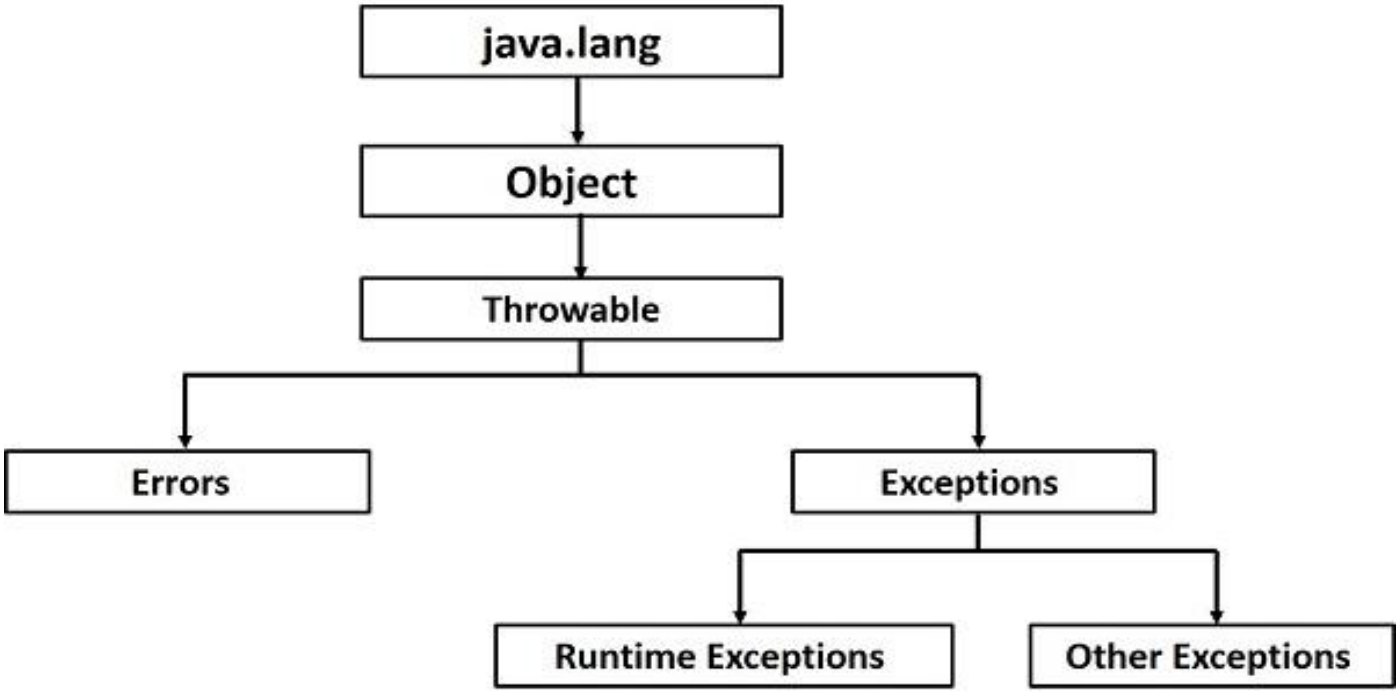
2. Unchecked Exceptions:

The exceptions which are raised at run time and checked by the JVM are called as 'Unchecked Exceptions'.

Exception Handling

Exception Hierarchy:

The java.lang package contains a class called 'Throwable'. This class consists of two sub classes namely 'Error' and 'Exception'. This Exception class contains all types of built-in exceptions. The following is the hierarchy of Exception class.



Exception Handling

```
class Ex
{
        public static void main(String args[])
        {
                System.out.println("Number of command line arguments passed:");
                int n=args.length;
                int div=46/n;
                System.out.println(n);
        }
}
```

The main aim of the above program is to display number of arguments passed to the array args[]. If we do not pass any arguments to args[] then ‘n’ will be zero. Now, it is not possible to divide ‘46’ with zero then it raises an arithmetic exception.

Exception Handling

The process of handling the exception is called 'Exception Handling'. When there is an exception, the user data may be corrupted. So, the exception should be handled by the programmer.

In Java, there are some keywords like try, catch, finally, throw and throws which are useful in the process of handling the exception.

Exception Handling: The try block

The programmer should observe the statements in his program where there may be a possibility of exception. Such statements are to be written inside a try block. A try block looks as follows:

```
try
{
        //statements;
}
```

If an exception is raised then the try block stores the exception details in an exception stack and then jumps into a catch block.

Exception Handling: The catch block

The programmer should write the catch block where the exception details to be displayed to the user. Here, the programmer can also display a message regarding what can be done to avoid the error. The catch block looks like as follows:

```
catch(ExceptionClass reference)
{
        //statements;
}
```

In the above structure, reference is automatically adjusted to refer the exception stack where the exception details are available.

Exception Handling: The catch block

We can display exception details by using any one of the following ways:

- Using the statement, **System.out.println(reference);**

- Using the method, **printStackTrace();** This method is present in Throwable class, which fetches the exception details from exception stack and display them.

Exception Handling: The finally block

The statements which are placed in the finally block are executed irrespective of whether there is an exception or not. The finally block looks as follows:

```
finally
{
        //statements;
}
```

Program to demonstrate try, catch and finally

Exception Handling: multiple catch clauses

Most of the time there is a possibility of more than one exception present in the program. In this case the programmer should write the multiple catch blocks to handle each one of them.

```
try
{
        //statements;
}
catch(ExceptionClass1 reference)
{
        //statements;
}
catch(ExceptionClass2 reference)
{
        //statements;
}
```

//Refer Program

Exception Handling: nested try block

If a try block is written within another try block that is called as 'Nested try block'. It looks like as follows:

```
try
{
        //statements;
        try
        {       //statements;             //Refer Program
        }
        catch(InnerTryExceptionClass reference)
        {       //statements;
        }
}
catch(OuterTryExceptionClass reference)
{       //statements;
}
```

Exception Handling: points to remember

- An exception can be handled using try, catch and finally blocks.
- It is possible to handle multiple exceptions using multiple catch blocks.
- Even though there is a possibility of several exceptions in try block, at a time only one exception will be raised.
- A single try block can be followed by several catch blocks.

Exception Handling: points to remember

- We cannot write a catch without try block, but we can write try block without any catch block.

- It is not possible to insert some statements between try and catch blocks.

- It is possible to write a try block within another try block called as ‘Nested try block’.

Exception Handling: The throw clause

Throw is used to throw an exception explicitly and then we can catch it.

In the following program we are creating an object of NullPointerException and throwing it explicitly as below:

throw new NullPointerException("Message to be displayed");

Now we can catch it by using the catch block as follows:

```
catch(NullPointerException npe)
{
        //statements;
}
```

//Refer Program

Exception Handling: The throws clause

In the case the programmer does not want to handle the checked exceptions, he should throw them out using 'throws' clause. Otherwise, an error will be raised by the compiler.

We have to use the throw clause with the method which is reason for the checked exception. We can use it as follows:

return_data_type method_name() throws Exception_class

Example:

void accept() throws IOException

public static void main(String args[])throws NullPointerException

Refer Program

Exception Handling: user-defined Exception

The exceptions which are created by the user and thrown by the user are known as 'Custom Exceptions' or 'User-defined Exceptions'.

Although Java' built-in exceptions handle most common errors. When we are developing an application in java, we often feel a need to create and throw our own exceptions. In that situation Java custom exceptions are used to customize the exception according to user need.

We can create a user-defined Exception class simply by extending Exception sub class Throwable.

Exception Handling: user-defined Exception

Steps to create user-defined exception:

1. User-defined exception class need to derived from the Exception class.

2. Override the constructor of Exception class to initialize a data member called ‘message’.

3. Override the public String toString() method of Exception class to return a message when an exception is raised.

4. Raise user-defined exception using ‘throw’ keyword whenever needed.

Refer Program

Exception Handling: Assertions

An assertion is a condition that should be true during the execution of the program. If the assertion condition is false, then it throws an Assertion Error. This makes the possibility to verify some expected condition is actually met or not. This can be done by using 'assert' keyword.

The 'assert' keyword has two forms. The first is shown below:

assert condition;

Here, condition is an expression that must evaluate to a boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default Assertion Error is displayed.

Exception Handling: Assertions

The second form is shown below:

assert condition:expr;

In this form, expr is a value that is passed to the Assertion Error constructor. This value is converted to its string format and displayed if an assertion fails.

Program for first form of assertion

Program for second form of assertion