[ Java – Core ]



|  |
| --- |
|  |

What is Java?



Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is:

* **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
* **Platform independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
* **Simple:** Java is designed to bke easy to learn. If you understand the basic concept of OOP Java would be easy to master.
* **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
* **Architectural-neutral:** Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
* **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
* **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
* **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
* **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
* **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
* **Distributed:** Java is designed for the distributed environment of the internet.
* **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:



James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called **Oak** after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later being renamed as **Java**, from a list of random words.

**Sun** released the first public implementation as Java 1.0 in **1995**. It promised **Write Once, Run Anywhere**(WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

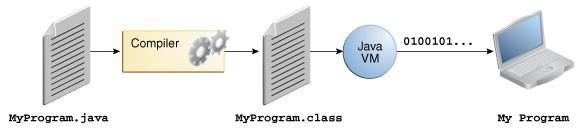
On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

# Process flow of execution



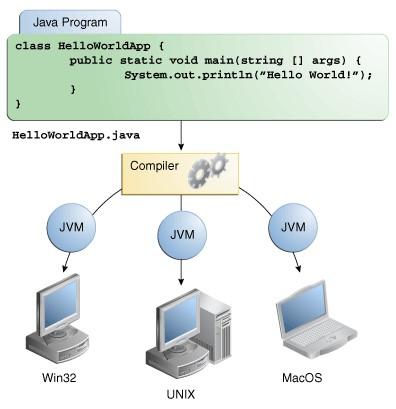
In the Java programming language, all source code is first written in plain text files ending with the .java extension.

Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine[1](https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html#FOOT) (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.



An overview of the software development process.

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS



Through the Java VM, the same application is capable of running on multiple platforms.

What is JVM?



JVM stands for ***Java Virtual Machine***. It is an abstract computing machine that is **responsible for executing Java programs**. When you write a Java program, the source code is compiled into byte code which is understandable by the JVM. Upon execution, the JVM translates byte code into machine code of the target operating system.

The JVM is the cornerstone of the Java programming language. It is responsible for the very well-known feature of Java: cross-platform. That means you can write a Java program once and run it anywhere: Windows, Linux, Mac and Solaris, as long as JRE is installed on the host operating system.

Every time you run a Java program, the JVM is started to execute and manage the program’s execution. The JVM is running in two modes: client (default) and server.

***An Oracle’s implementation for JVM is called Java HotSpot VM***



What is JRE?



JRE stands for ***Java Runtime Environment***. It provides the **libraries**, JVM and other components necessary for you to run applets and applications written in the Java programming language.

The JRE contains standard tools such as java, keytool, policytoo, but it doesn’t contain compilers or debuggers for developing applets and applications.

When you deploy your Java applications on client’s computer, the client needs a JRE to be installed.

What is JDK?



JDK stands for ***Java Development Kit***. It’s a superset of JRE. The JDK includes the JRE plus command-line development tools such as compilers (javac) and debuggers (jdb) and others (jar, javadoc, etc) that are necessary or useful for developing applets and applications.

Therefore, as a Java programmer, you have to install JDK as a minimum requirement for theh development environment.

Summary:

* **JVM** = JVM is the Virtual Machine that runs Java applications. The JVM makes Java platform independence
* **JRE** = JVM + standard libraries: provides environment for executing Java applications ▪ **JDK** = JRE + development tools for compiling and debugging Java applications

Tips:

* You should have both JRE and JDK installations (setup) on your computer. You will need both during the development process.
* You should have multiple versions of JDK and JRE installed: 1.5, 1.6, 1.7 and 1.8 for different testing purposes in the future.
* You should install both 32-bit and 64-bit versions.
* When installing the JDK, remember to check ‘Install Demos and Samples’. Then you can explore various interesting examples in the **demo** directory under JDK’s installation path.
* Only the JDK includes source code of the Java runtime libraries. You can discover the source code in the **src.zip** file which can be found under JDK’s installation directory.

First Java Program:



public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\*/

public static void main(String []args) {

System.out.println("Hello World"); // prints Hello World

}

}

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open notepad and add the code as above.
2. Save the file as: MyFirstJavaProgram.java.
3. Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
4. Type ' javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
5. Now, type ' java MyFirstJavaProgram ' to run your program.
6. You will be able to see ' Hello World ' printed on the window.

C:\> javac MyFirstJavaProgram.java

C:\> java MyFirstJavaProgram

Hello World

Basic Syntax:



About Java programs, it is very important to keep in mind the following points.

* **Case Sensitivity -** Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
* **Class Names -** For all class names the first letter should be in Upper Case.

If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. Example *class MyFirstJavaClass*

* **Method Names -** All method names should start with a Lower Case letter.

If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example *public void myMethodName()*

* **Program File Name -** Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match your program will not compile). Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*
* **public static void main(String args[]) -** Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers:



All Java components require names. Names used for classes, variables and methods are called identifiers.

In Java, there are several points to remember about identifiers. They are as follows:

* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character identifiers, can have any combination of characters.
* A keyword cannot be used as an identifier.
* Most importantly identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value
* Examples of illegal identifiers: 123abc, -salary

Java Keywords:



The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

| abstract | Assert | boolean | break |
| --- | --- | --- | --- |
| byte | case | catch | char |
| class | const | continue | default |
| do | double | else | enum |
| extends | Final | finally | float |
| for | Goto | if | implements |
| import | instanceof | int | interface |
| long | native | new | package |
| private | protected | public | return |
| short | Static | strictfp | super |
| switch | synchronized | this | throw |
| throws | transient | try | void |
| volatile | While |  |  |

# Comments in Java



Java supports single-line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

public class MyFirstJavaProgram{

/\* This is my first java program.

\* This will print 'Hello World' as the output \* This is an example of multi-line comments. \*/

public static void main(String []args){

// This is an example of single line comment

/\* This is also an

example of single line comment. \*/

System.out.println("Hello World");

}

}

# Datatypes



Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

* Primitive Data Types
* Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

The Java language has 8 primitive types: **boolean**, **byte**, **char**,**double**, **float**, **int**, **long**, and **short**.

A **boolean** type represents either *true* or *false* value.

A **char** type represents a single character, such as 'a', 'B', 'c', ...Actually char type is 16-bit integer number (un-signed).

The others are numeric types. The following table lists the primitive types which represent numbers in the Java language (the char type is also included because it is actually a number type):

|  | **Type** | **Bits** | **Bytes** | **Minimum value** | **Maximum value** |
| --- | --- | --- | --- | --- | --- |
| ***Integer numbers*** | **byte** | 8 | 1 | **-128** (-27) | **127** (27-1) |
| **char** | 16 | 2 | **0** | **65,535** |
| **short** | 16 | 2 | **-32,768** (-215) | **32,767** (215-1) |
| **int** | 32 | 4 | **-2,147,483,648** (-231) | **2,147,483,647** (231-1) |
| **long** | 64 | 8 | approx. -9,2 billions of billion (-  263) | approx. 9,2 billions of billion (-2631) |
| ***Floating point***  ***numbers*** | **float** | 32 | 4 | N/A | N/A |
| **Double** | 64 | 8 | N/A | N/A |

Reference Data Types:

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
* Class objects, and various type of array variables come under reference data type.
* Default value of any reference variable is null.
* A reference variable can be used to refer to any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

# Variable Types



A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. The basic form of a variable declaration is shown here:

data type variable [ = value][, variable [= value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java: int a, b, c; // Declares three ints, a, b, and c.

int a = 10, b = 10; // Example of initialization byte B = 22; // initializes a byte type variable B. double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a'; // the char variable a iis initialized with value 'a'

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

Local variables:



* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
* Access modifiers cannot be used for local variables.
* Local variables are visible only within the declared method, constructor or block.
* Local variables are implemented at stack level internally.
* There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

public class Test{ public void pupAge(){ int age = 0; age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]){ Test test = new Test(); test.pupAge();

} }

| **Local Variable Demo 1** |
| --- |
| public class LocalVarDemo\_2  {  public static void main(String[] args) { |

Program N0. 2

| }  } | int age=21; //local variables  System.out.println("\n\t Value of age = " + age); |
| --- | --- |

| **Local Variable Demo 2** |
| --- |
| public class LocalVarDemo\_3  {  public static void main(String[] args)  {  char ch; float avg; String str;    ch = 'A'; avg = 45.67f;  str = "Welcome";    System.out.println("\n\t Value of ch = " + ch);  System.out.println("\n\t Value of avg = " + avg); System.out.println("\n\t Value of str = " + str); }  } |

Program N0. 3

| **Local Variable Demo 3** |
| --- |
| public class LocalVarDemo\_4  {  static void showvalue()  {  int age = 32; //local variable  System.out.println("\n\t Value of age = " + age);  }  public static void main(String[] args)  {  showvalue();  }  } |

Program N0. 4

Instance variables:



* Instance variables are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, a slot for each instance variable value is created.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
* Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
* Instance variables can be declared in class level before or after use.
* Access modifiers can be given for instance variables.
* The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
* Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
* Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class ( when instance variables are given accessibility) should be called using the fully qualified name . *ObjectReference.VariableName*.

Example:

import java.io.\*;

public class Employee{

// this instance variable is visible for any child class. public String name;

// salary variable is visible in Employee class only. private double salary;

// The name variable is assigned in the constructor. public Employee (String empName){ name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal){ salary = empSal;

}

// This method prints the employee details. public void printEmp(){

System.out.println("name : " + name );

System.out.println("salary :" + salary);

}

public static void main(String args[]){ Employee empOne = new Employee("Ransika"); empOne.setSalary(1000); empOne.printEmp();

} }

| **Instance Variable Class Student** |
| --- |
| class Student  {  public int roll; // instance varialbe  String name; // data members / fields    public void showStud()  {  System.out.println("\n\t Roll No :" + roll);  System.out.println("\n\t Name :" + name);  }  }  public class InstanceVarDemo\_5  {  public static void main(String[] args)  {  Student stud = new Student();  stud.roll = 101; stud.name = "Chetan";  stud.showStud();  }  } |

Program N0. 5

| **Instance Variable Default Values** |
| --- |
| class Demo  {  int i; byte b; |

Program N0. 6

| String str; boolean bl;  }  public class InstanceVarDemo\_6  {  public static void main(String[] args)  {  Demo d1 = new Demo();  System.out.println("Default value of int i = " + d1.i);  System.out.println("Default value of byte b = " + d1.b);  System.out.println("Default value of String str = " + d1.str); System.out.println("Default value of boolean bl = " + d1.bl); }  } |
| --- |

| **Class marketing dept** |
| --- |
| class MktDept  { int empid; int sal;    static String deptName;  }  public class StaticVarDemo\_7  {  public static void main(String[] args)  {  //MktDept m1 = new MktDept();  //m1.empid = 101;    MktDept.deptName = "Marketing Department";  System.out.println("\n\t Dept name = " + MktDept.deptName);  }  } |

Program N0. 7

# Assignment



| **Class Product** |
| --- |
|  |

Program N0. 8

Class/static variables:



* Class variables also known as static variables are declared with the ***static*** keyword in a class, but outside a method, constructor or a block.
* There would only be one copy of each class variable per class, regardless of how many objects are created from it.
* Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
* Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
* Static variables are created when the program starts and destroyed when the program stops.
* Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
* Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
* Static variables can be accessed by calling with the class name*ClassName.VariableName*.
* When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

public class Employee{

// salary variable is a private static variable private static double salary; // DEPARTMENT is a constant public static final String DEPARTMENT = "Development "; public static void main(String args[]){ salary = 1000;

System.out.println(DEPARTMENT + "average salary:" + salary);

}

}

**Note:** If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

# Java Modifier



* Modifiers are keywords that you add to those definitions to change their meanings.
* The Java language has a wide variety of modifiers, including the following: o Java Access Modifiers o Non Access Modifiers

# Java Access Modifiers



Default Access Modifier

* Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
* A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default

public.

| **Class Student** |
| --- |
| class Student  {  int sid; //default members  String sname;    void showStud()  {  System.out.println("\n\t Roll No :" + sid);  System.out.println("\n\t Name :" + sname);  } }  public class DefaultAccessDemo\_9  {  public static void main(String[] args)  {  Student s1 = new Student();  s1.sid = 101; |

Program N0. 9

|  |  | s1.sname = "Prakash"; s1.showStud(); |
| --- | --- | --- |
| } | } |  |

## Public Access Modifier

* A class, method, constructor, interface etc declared public can be accessed from any other class.
* Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

| **Class Employee** |
| --- |
| class Employee  { public int empid; public String ename;    private void showEmp()  {  System.out.println("\n\t Empid :" + empid);  System.out.println("\n\t Emp Name :" + ename); }  }  public class PublicModifierDemo  {  public static void main(String[] args)  {  Employee e1 = new Employee();  e1.empid = 101; e1.ename = "Sneha";  e1.showEmp();  }  } |

**Program N0.**

**10**

Private Access Modifier

* Methods, Variables and Constructors that are declared private can only be accessed within the declared class

itself

* Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

Protected Access Modifier

* Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

# Java Non Access Modifiers



Static Variables:

* The static key word is used to create variables that will exist independently of any instances created for the class.
* Only one copy of the static variable exists regardless of the number of instances of the class

| **Static Variable Demo** |
| --- |
| class Demo  {  int x, y; static int count; |

🞆 Static variables are also known as class variables. Local variables cannot be declared static **Program N0. 11**

| void setxy()  {  X = 1; Y = 1; count++;  } }  public class StaticDemo1  {  public static void main(String[] args)  {  System.out.println(“\n\t No. of Objects = “ +  Demo.count);    Demo d1 = new Demo(); d1.setxy();    Demo d2 = new Demo(); d2.setxy();    System.out.println(“\n\t No. of Objects = “ +  Demo.count);  }  } |
| --- |

Static Methods

* The static key word is used to create methods that will exist independently of any instances created for the class
* Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables
* Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method

| **Class Product (Static data members and static methods)** |
| --- |
| class Product  { int pid, price, qty; static double taxRate;    static void setTax()  {  pid = 101;  taxRate = 5.00;  }  static double getTax()  {  return taxRate;  } }  public class StaticDemo2  {  public static void main(String[] args)  {  Product.setTax();  System.out.println("Tax = " + Product.getTax());  }  } |

**Program N0.**

**12**

# 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 a different object
* However the data within the object can be changed. So the state of the object can be changed but not the reference

| **Class Circle** |
| --- |
| class Circle  {  double rad;  final double PI=3.14;    void setRadius(double r)  {  rad = r;  }  double getArea()  {  return PI\*rad\*rad;  }  }  public class FinalVarDemo1\_13  {  public static void main(String[] args)  {  Circle cir1 = new Circle(); cir1.setRadius(4.30);  System.out.println("\n\t Area of cir1 = " + cir1.getArea());    }  } |

**Program N0.**

**13**

| **Final Object** |
| --- |
| public class FinalVarDemo\_14  {  public static void main(String[] args)  {  String str1 = "Welcome";    System.out.println("\n\t str1 = " + str1);    final String str2 = "Hello World";    System.out.println("\n\t str2 = " + str2);    str1 = "Welcome to Java";  System.out.println("\n\t str1 = " + str1);    str2 = "This is New String";  System.out.println("\n\t str2 = " + str2);  }  } |

**Program N0.**

**14**

## 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

## Final Classes

▪ The main purpose of using a class being declared as final is to prevent the class from being sub classed ▪ If a class is marked as final then no class can inherit any feature from the final class

# Scanner Class



* Scanner class is defined in **java.util** package, it used to perform input operations
* The Scanner class breaks the input into tokens using a delimiter which is whitespace by default ▪ It provides many methods to read and parse various primitive values Commonly used methods of Scanner class:

| **Method** | **Description** |
| --- | --- |
| **pubic String next()** | Returns the next token from the scanner |
| **public String nextLine()** | Moves the scanner position to the next line and returns the value as a string. |
| **public byte nextByte()** | Scans the next token as a byte |
| **public short nextShort()** | Scans the next token as a short value |
| **public int nextInt()** | Scans the next token as an int value |
| **public long nextLong()** | Scans the next token as a long value |
| **public float nextFloat()** | Scans the next token as a float value |
| **public double nextDouble()** | Scans the next token as a double value |

| **Input Age using Scanner Class** |
| --- |
| import java.util.Scanner; public class ScannerDemo1\_15  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);    int age;    System.out.print("\n\t Enter your Age :");  age = scan.nextInt();    System.out.println("\n\t Your Age :" + age); }  } |

**Program N0.**

**15**

| **Input average marks, name, character using scanner class** |
| --- |
| import java.util.Scanner; public class ScannerDemo2\_16  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in); |

**Program N0. 16**

| }  } | float avg; char ch; String sname;  System.out.print("\n\t Enter Average Marks :"); avg = scan.nextFloat();  System.out.print("\n\t Enter Any Character :"); ch = scan.next().charAt(0);  System.out.print("\n\t Enter Your Name :"); sname = scan.next();  System.out.println("\n\t Average Marks :" + avg);  System.out.println("\n\t Character :" + ch);  System.out.println("\n\t Name :" + sname); |
| --- | --- |

# Assignment



**Program N0. 17:** Sum of two numbers

**Program N0. 18:** Input roll no, name marks of 3 subjects and calculate total marks, Average marks of a student **Program N0. 19:** Input radius of a circle and calculate area and circumference

# Command Line Arguments



* A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.
* The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run.
* For example, suppose a Java application called Sort sorts lines in a file. To sort the data in a file named friends.txt, a user would enter: **java Sort friends.txt**
* When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings

| **Command line arguments display length of arguments** |
| --- |
| public class CmdLIneArgsDemo1  {  public static void main(String[] a)  {  System.out.println("\n\t No. of arguments :" + a.length); }  } |

**Program N0.**

**20**

| **Command line arguments – pass two arguments and display the same** |
| --- |

**Program N0. 21**

public class CmdLIneArgsDemo2

{

public static void main(String[] a)

{

System.out.println("\n\t No. of arguments :" + a.length);

System.out.println("\n\t 1st Argument :" + a[0]);

System.out.println("\n\t 2nd Argument :" + a[1]);

} }

| **Command line arguments – display all arguments using loop** |
| --- |
| public class CmdLIneArgsDemo3  {  public static void main(String[] a)  {  System.out.println("\n\t No. of arguments :" + a.length);    for(int i=0; i<a.length; i++)  {  System.out.println("\n\t a["+i+"]:" + a[i]);  }  }  } |

**Program N0.**

**22**

▪ If an application needs to support a numeric command-line argument, it must convert a String argument that represents a number, such as "34", to a numeric value.

| **Parsing Numeric Command-Line Arguments** |
| --- |
| public class CmdLIneArgsDemo4  {  public static void main(String[] args)  {  int n1, n2;  n1 = Integer.parseInt(args[0]); n2 = Integer.parseInt(args[1]);    int sum = n1+n2;    System.out.println("\n\t n1 = " + n1);  System.out.println("\n\t n2 = " + n2);  System.out.println("\n\t sum = " + sum);  }  } |

**Program N0.**

**23**

# Decision Making Statements



* The program executes instructions sequentially. Sometimes, a program requires checking of certain conditions in program execution
* Java provides various conditional statements to check condition and execute statements according conditional

criteria

* Followings are the different conditional statements used in Java
* If Statement
* If-Else Statement
* Nested If-Else Statement
* Switch Case

## The if Statement

* An if statement consists of a Boolean expression followed by one or more statements
* If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed. **The syntax of an if statement is:**

if(Boolean\_expression)

{

//Statements will execute if the Boolean expression is true

}

| **Input two numbers and display the max** |
| --- |
| import java.util.Scanner; public class IfDemo1  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int a, b;    System.out.print("\n\t Enter a :");   1. = scan.nextInt();     System.out.print("\n\t Enter b :");   1. = scan.nextInt();     if(a>b)  System.out.println("\n\t a is Max");    if(b>a)  System.out.println("\n\t b is Max");  }  } |

**Program N0.**

**24**

| **Input Average marks and display the result pass if avg is greater than or equal to 35, otherwise result is fail** |
| --- |
| import java.util.Scanner; public class IfDemo2  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  float avg;    System.out.print("\n\t Enter Average Marks :");  avg = scan.nextFloat();    if(avg>=35)  {  System.out.println("\n\t You Are Pass");  }  else  {  System.out.println("\n\t You Are Fail");  }  }  } |

**Program N0.**

**25**

| **Input a number and display whether it is positive, negative or zero** |
| --- |
| import java.util.Scanner; public class IfDemo3  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int no;    System.out.print("\n\t Enter a no. :");  no = scan.nextInt();    if(no>0)  System.out.println("\n\t number is Positive"); else if(no==0)  System.out.println("\n\t number is Zero");  else  System.out.println("\n\t number is Negative");  }  } |

**Program**

### **N0. 26**

| **Input Average marks and display the appropriate grades** |
| --- |
| import java.util.Scanner; public class IFDemo4\_27  {  public static void main(String[] args)  {  Scanner sc = new Scanner(System.in);  float avg;    System.out.print("\n\t Enter average marks :");  avg = sc.nextFloat();    if(avg>=75)  System.out.println("\n\t Distinction"); else if(avg>=60)  System.out.println("\n\t First Class"); else if(avg>=45)  System.out.println("\n\t Second Class");  else if(avg>=35)  System.out.println("\n\t Pass");  else  System.out.println("\n\t Fail");  }  } |

**Program**

### **N0. 27**

| **Input gender and age of an employee. An employee is eligible for insurance in following conditions**   1. **Employee is male and age is greater than 30** 2. **Employee is Female and age is greater that 25** |
| --- |
| import java.util.Scanner; public class IFDemo5\_28 {  public static void main(String[] args){  Scanner sc = new Scanner(System.in);  String gender;  int age;    System.out.print("\n\t Enter Gender :");  gender = sc.next();    System.out.print("\n\t Enter Age :");  age = sc.nextInt();    if(gender.toLowerCase().equals("male"))  { |

**Program**

### **N0. 28**

if(age>=30)

System.out.println("\n\t Employee is Eligible for

Insurance");

else

System.out.println("\n\t Not Eligible");

}

else if(gender.equalsIgnoreCase("female"))

{

if(age>=25)

System.out.println("\n\t Employee is Eligible for

Insurance");

else

System.out.println("\n\t Not Eligible");

}

else

{

System.out.println("\n\t Wrong input");

}

}

}

# Assignments



**Input price and qty, calculate bill amount with discount of 5.00% if price is greater than or equal**

**Program N0. 29** **to 50**

**Program N0. 30** **Input a number and display whether it is Odd or Even**

**Program N0. 31** **Input an alphabet and check whether it is vowel or consonant**

# Logical Operators



Logical AND (&&)

* This operator is used to evaluate 2 or more conditions or expressions with relational operators simultaneously. If all expressions to the left and right of the logical expression are TRUE then the whole compound expression is TRUE

| **Input three numbers and display max** |
| --- |
| import java.util.Scanner;    public class IFDemo5\_28  {  public static void main(String[] args)  {  Scanner sc = new Scanner(System.in);    int a, b, c;    System.out.print("\n\t Enter a :");   1. = sc.nextInt();     System.out.print("\n\t Enter b :");   1. = sc.nextInt();     System.out.print("\n\t Enter c :");   1. = sc.nextInt();     if(a>b && a>c)  {  System.out.println("\n\t a is max");  }  else if(b>a && b>c) |

**Program**

**N0. 32**

|  |  | {  System.out.println("\n\t b is max");  } else  {  System.out.println("\n\t c is max"); } |
| --- | --- | --- |
| } | } |  |

Logical OR(||)

* This operator is used to evaluate or combine 2 or more expressions or conditions, and evaluates to TRUE if any one of all the expressions is true

| **Input any alphabet and display whether it is a vowel or a consonant** |
| --- |
| import java.util.Scanner;    public class IfDemo7  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  char ch;  System.out.print("\n\t Enter any alphabet :");  ch = scan.next().charAt(0);    if(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u') System.out.println("\n\t It is a Vowel");  else  System.out.println("\n\t It is Consonant");  }  } |

**Program**

**N0. 33**

# Switch Statement



* The Switch Case Statement is used to make a decision from the number of choices
* The expression following the keyword switch is any C expression that evaluates an integer or a char value
* First, the expression following the keyword switch is evaluated. The value is then matched, one by one, against the constant values that follow the case statements. When a match is found, the program executes the statements following that case, and all subsequent case and default statements as well.
* If no match is found with any of the case statements, only the statements following the default are executed. A few examples will show how this control structure works.
* Expressions can also be used in cases provided they are constant expressions. Thus **case 3 + 7** is correct, however, **case a + b** is incorrect
* The **break** statement when used in a **switch** takes the control outside the **switch**. However, use of **continue** will not take the control to the beginning of **switch** as one is likely to believe

| **Switch Demo** |
| --- |
| public class SwitchDemo1 {  public static void main(String[] args) {  int no; no = 2;  switch(no)  {  case 1:  System.out.println("\n\t It is Number 1"); |

**Program N0. 34**

| break; case 2:  System.out.println("\n\t It is Number 2");  break; default:  System.out.println("Invalid");  }  }  } |
| --- |
| **Addition..Multiplication.. Menu Program** |
| import java.util.Scanner; public class SwitchDemo2 { public static void main(String[] args) { Scanner scan = new Scanner(System.in); int num1, num2, ans,choice;  System.out.print("\n\t Enter Num1 :"); num1 = scan.nextInt();  System.out.print("\n\t Enter Num2 :");  num2 = scan.nextInt();  System.out.println("\n\t\t1. Addition \n\t\t 2. Subtraction  \n\t\t 3. Multiplication \n\t\t 4. Division"); System.out.print("\n\t Enter a choice : ");  choice = scan.nextInt(); switch(choice)  {  case 1:  ans = num1+num2;  System.out.println("\n\t Addition = " + ans);  break; case 2:  ans = num1-num2;  System.out.println("\n\t Subtraction = " + ans);  break; case 3:  ans = num1\*num2;  System.out.println("\n\t Multiplication = " + ans);  break; case 4:  ans = num1/num2;  System.out.println("\n\t Division = " + ans);  break; default:  System.out.println("\n\t Wrong Input");  }  }  } |

Program N0.

35

| **Pizza Order** |
| --- |
| import java.util.Scanner; public class SwitchDemo3 {  public static void main(String[] args) { Scanner scan = new Scanner(System.in);  char pizzasize; |

Program N0.

36

|  |  | System.out.println("\n\t Enter Pizza Size[s/m/l]:"); pizzasize = scan.next().charAt(0);    switch(pizzasize)  { case 's':  System.out.println("\n\t Pizza size : Small");  break; case 'm':  System.out.println("\n\t Pizza size : Medium");  break; case 'l':  System.out.println("\n\t Pizza size : Large");  break; default:  System.out.println("\n\t Invalid Input"); } |
| --- | --- | --- |
| } | } |  |

# Assignments



| **Program No. 37** | If cost price and selling price of an item is input through the keyboard, write a program to determine whether the seller has made profit or incurred loss. Also determine how much profit he made or loss he incurred. | |
| --- | --- | --- |
| **Program No. 38** | Any year is input through the keyboard. Write a program to determine whether the year is a leap year or not. (Hint: Use the % (modulus) | |
| **Program No. 39** | Write a program to check whether a triangle is valid or not, when the three angles of the triangle are entered through the keyboard. A triangle is valid if the sum of all the three angles is equal to 180 degrees | |
| **Program No. 40** | A certain grade of steel is graded according to the following conditions:   1. Hardness must be greater than 50 2. Carbon content must be less than 0.7 (iii) Tensile strength must be greater than 5600 The grades are as follows:   Grade is 10 if all three conditions are met  Grade is 9 if conditions (i) and (ii) are met  Grade is 8 if conditions (ii) and (iii) are met  Grade is 7 if conditions (i) and (iii) are met  Grade is 6 if only one condition is met  Grade is 5 if none of the conditions are met  Write a program, which will require the user to give values of hardness, carbon content and tensile strength of the steel under consideration and output the grade of the steel | |
| **Program No. 41** | A library charges a fine for every book returned late. For first 5 days the fine is 50 paise, for 6-10 days fine is one rupee and above 10 days fine is 5 rupees. If you return the book after 30 days your membership will be cancelled. Write a program to accept the number of days the member is late to return the book and display the fine or the appropriate message. | |
| **Program No. 42** | The policy followed by a company to process customer orders is given by the following rules:   1. If a customer order is less than or equal to that in stock and has credit is OK, supply has requirement. 2. If has credit is not OK do not supply. Send him intimation. | |
|  | c. | If has credit is Ok but the item in stock is less than has order, supply what is in stock.  Intimate to him data the balance will be shipped.  Write a C program to implement the company policy. |

# Java Loops



* There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.
* Java has very flexible three looping mechanisms. You can use one of the following three loops:
  + while Loop
  + do...while Loop
  + for Loop
  + As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

# The while Loop



| **Syntax:**  **while(Boolean\_expression)**  **{**  **//Statements**  **}** |
| --- |

* A while loop is a control structure that allows you to repeat a task a certain number of times.
* When executing, if the boolean\_expression result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.
* Here, key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

| **Print helloworld 5 times** |
| --- |
| public class WhileLoopDemo1 {  public static void main(String[] args) {  int i;    i = 1;  while(i<=5)  {  System.out.println("\n\t Hello World"); i++;  }  }  } |

**Program N0.**

**43**

| **Input 10 Number using While loop** |
| --- |
| import java.util.Scanner; public class WhileDemo\_44  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);    int no, i; i = 1;  while(i<=10)  { |

**Program N0. 44**

|  |  | } | System.out.print("\n\t Enter no :"); no = scan.nextInt(); i++; |
| --- | --- | --- | --- |
| } | } |  |  |

| **Input 10 numbers and display count of positive, negative, zeros** |
| --- |
| import java.util.Scanner; public class WhileDemo\_45  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in); int no, i, pos=0, neg=0, zero=0;  i = 1;    while(i<=10)  {    System.out.print("\n\t Enter no :");  no = scan.nextInt();  if(no>0)  pos++;  else if(no==0)  zero++; else neg++;  i++;  }  System.out.println("\n\t Total Positives :" + pos);  System.out.println("\n\t Total Negatives :" + neg);  System.out.println("\n\t Total Zeros :" + zero);  }  } |

**Program N0.**

**45**

# do...while Loop



* A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time
* Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

| **Syntax:**  **Do**  **{**  **//Statements**  **} while(Boolean\_expression);** |
| --- |

* If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

▪

| **Do..while demo** |
| --- |
| public class DoWhileLoopDemo1  { public static void main(String[] args) { int i = 10;    do  {  System.out.println("Value of i = " + i); |

**Program N0. 46**

|  |  | i++;  }while(i<=5); |
| --- | --- | --- |
| } | } |  |

7

| **Input Record of Student until user enters “stop”** |
| --- |
| import java.util.Scanner;    public class WhileDemo\_47  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int roll;  choice;    do  {  System.out.print("\n\n\t Enter Roll No. :");  roll = scan.nextInt();    System.out.print("\n\n\t Enter Name :");  name = scan.next();    System.out.print("\n\n\t Next Record? [ok=Next |stop = Cancel] :" );  choice = scan.next();    }while(!choice.equalsIgnoreCase("stop"));  }  } |

**Program N0.**

**47**

# For loop



| **Syntax**: for(initialization; Boolean\_expression; update)  {  //Statements } |
| --- |

* A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
* A for loop is useful when you know how many times a task is to be repeated.
* The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
* Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
* After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.

| **For Loop Demo** |
| --- |
| public class ForDemo\_48  {  public static void main(String[] args)  {  for(int i=0; i<5; i++)  { |

* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates **Program N0. 48**

|  |  | } | System.out.println("i = " + i); |
| --- | --- | --- | --- |
| } | } |  |  |

## Enhanced for loop in Java



| **Syntax:**  for(declaration : expression)  {  //Statements  } |
| --- |

* As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.
* **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
* **Expression:** This evaluates to the array you need to loop through.

The expression can be an array variable or method call that returns an array.

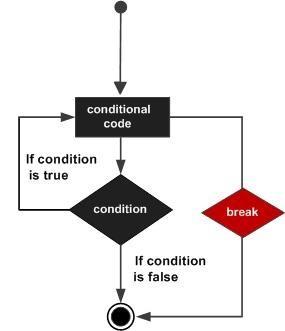
| **Enhanced For loop Demo** |
| --- |
| public class ForDemo\_49{  public static void main(String[] args)  {  System.out.println("\n\t Command line arguments are :");    for(String item : args)  {  System.out.println(item);  }  }  } |

**Program N0.**

**49**

### The break Keyword

▪ The break keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch



statement.

| **Break Statement** |
| --- |
|  |

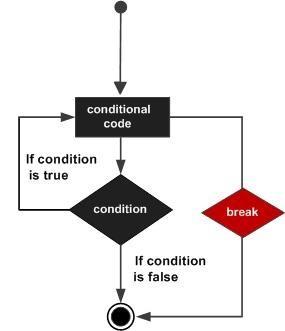
Program N0.

50

## The continue Keyword



▪ The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop



| **Continue Statement** |
| --- |
| /\* Java49. WAP to display the numbers upto 100 by using continue keyword to continue or break \*/    import java.util.Scanner; class Java49  {  public static void main(String []args)  {  Scanner scan = new Scanner(System.in);    int cnt,i;  String result;    cnt=0;  for(i=1;i<=100;i++)  {  System.out.print(" " + i);  cnt++;  if(cnt==10)  {  cnt=0;  System.out.println("\n\t Do you to Continue  [yes/no] :");  result = scan.next();    if(result.equals("yes"))  {  continue;  } else  {  break;  }  }  }  }  } |

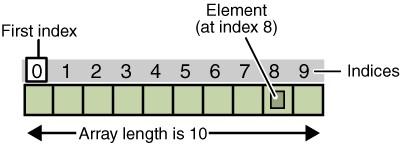
Program N0.

51

# Array



* java provides a data structure Array, which stores a fixed-size sequential collection of elements of the same type
* An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type
* Each item in an array is called an element, and each element is accessed by its numerical index. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.



characteristics of arrays in Java:

* Fixed length: Once an array is created, we cannot change its size. So consider using arrays when the numbers of elements are known and fixed. Otherwise, you should consider using another dynamic container such as ArrayList.
* Fast access: It’s very fast to access any elements in an array (by index of the elements) in constant time:

accessing the 1st element takes same time as accessing the last element. So performance is another factor when choosing arrays.

* An array can hold primitives or objects.
* An array of primitives stores values of the primitives.
* An array of objects stores only the references to the objects.
* In Java, the position of an element is specified by index which is zero-based. That means the first element is at index 0, the second element at index 1, and so on.
* An array itself is actually an object.

## Declaring a Variable to Refer to an Array ▪ Declare an Array

<DataType>[] ArrVarName;

* Like declarations for variables of other types, an array declaration has two components: the Data Type and the array's name.
* An array's type is written as type[], where type is the data type of the contained elements ▪ The size of the array is not part of its type (which is why the brackets are empty).
* An array's name can be anything you want, provided that it follows the rules and conventions
* It simply tells the compiler that this variable will hold an array of the specified type
* Example o byte[] arrayRefVar; o short[] arrayRefVar; o long[] arrayRefVar; o float[] arrayRefVar; o double[] arrayRefVar;

o boolean[] arrayRefVar; o char[] arrayRefVar; o String[] arrayRefVar;

## Creating Arrays

* You can create an array by using the new operator with the following syntax: arrayRefVar = new dataType[arraySize];
* The above statement does two things:
* It creates an array using new dataType[arraySize];
* It assigns the reference of the newly created array to the variable arrayRefVar
* Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

dataType[] arrayRefVar = new dataType[arraySize];

## Arrays declarations and initialization

Declare first and initialize later:

int [] numbers = new int[10];

This declares an array object to hold 10 integer numbers (primitive array).

When declaring an array of primitive type, remember these rules:

▪ All numbers are initialized to zeroes by default. That means the above array numbers contain 10 numbers which are all zeroes, even we haven’t initialized the array yet. ▪ Boolean elements are initialized to false by default.

Then we initialize values for each element of the array like this:

numbers[0] = 10;

numbers[1] = 500; numbers[2] = 1000; …

The following statement declares an array of String objects:

String[] names = new String[5];

This array holds 5 String objects. And by default, all elements of Object type are initialized to null.

**NOTE:** In Java, you can place the brackets **[]** either after the type or after the variable name. Hence these declarations are both correct:

String[] names = new String[5];

String titles[] = new String[10];

However, it’s recommended to use the **[]** after the type for readability: You can easily realize this is an array of Strings, or that is an array of integer numbers.

You can also declare and initialize elements of an array in one statement. For example:

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

This creates an array with 10 integer numbers which are initialized up declaration. This is a handy shortcut for declaring arrays whose elements are already known at compile time.

This statement declares and initializes an array of Strings:

String[] columnNames = {“No”, “Name”, “Email”, “Address”};

**Example:**

▪ Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

double[] myList = new double[10];

# Arrays Manipulation



We access elements in the array by index (remember 0-based):

String firstColumn = columnNames[0];

This statement takes value of the first element in the String array columnNames and assigns it to the variable firstColumn.

The following statement illustrates accessing an element in a 2D array:

String[] firstRow = sampleData[0];

This gets the first element in the sampleData array, which returns an array.

The following statement takes the element at 3rd row and 2nd column in the above 2D array:

String letter = sampleData[4][3];

And these examples show how to assign values to elements in arrays:

columnNames[2] = “Phone;

columnNames[4] = new String(“City”); numbers[4] = 1024; sampleData[4][2] = “xyz”;

A common operation is iterating an array using a loop statement like the for statement. The following example uses the for loop to iterate over all elements in an array of integer numbers, and prints value of each element:

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int i = 0; i < numbers.length; i++) {

System.out.println(numbers[i]); }

For arrays of objects you can use the *for each* syntax to iterate. For example:

String[] nameList = {"Tom", "Mary", "Peter", "John", "Adam", "Justin"};

for (String aName : nameList) {

System.out.println(aName);

}

| **Array Demo** |
| --- |
| public class ArrayDemo1\_52  {  public static void main(String[] args)  {  int[] arr; //declaration of array reference variable arr = new int[5]; //creating array of 5 elements  int[] brr; arr[0] = 10; arr[1] = 20; arr[2] = 30; arr[3] = 40; arr[4] = 50;  System.out.println("\n\t 1st Value = " + arr[0]);  }  } |

Program N0.

52

| **Input an array of 5 elements and display using for/foreach loop** |
| --- |
| import java.util.Scanner; public class ArrayDemo2\_53  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int[] arr = new int[5];  System.out.println("\n\t Length of arr = " + arr.length); for(int i=0; i<arr.length; i++)  {  System.out.print("\n\t Enter Value :");  arr[i] = scan.nextInt();  }  System.out.println("\n\t Array Values are :");  for(int i=0; i<arr.length; i++)  {  System.out.print("\t" + arr[i]);  }  }  } |

Program N0.

53

| **Input an array of 5 elements and display all odd and even numbers** |
| --- |
| import java.util.Scanner; class Java54  {  public static void main(String []args)  {  Scanner scan = new Scanner(System.in);    int size,i;  System.out.print("\n\t Enter Size of Array :");  size = scan.nextInt(); int arr[] = new int[size];  System.out.print("\n\t Enter "+ size +" Array Values :"); for(i=0;i<size;i++)  { |

Program N0.

54

|  |  | arr[i] = scan.nextInt();  }  System.out.print("\n\t Even Array values are :"); for(i=0;i<size;i++)  {  if(arr[i]%2==0)  System.out.print(" " + arr[i]);  }  System.out.print("\n\t Odd Array values are :"); for(i=0;i<size;i++)  {  if(arr[i]%2==1)  System.out.print(" " + arr[i]);  } |
| --- | --- | --- |
| } | } |  |

| **Input an array of 5 elements and display all max and min numbers** |
| --- |
| import java.util.Scanner; class Java55  {  static Scanner scan;  public static void main(String []args)  {  scan = new Scanner(System.in);  int size,arr[];    System.out.print("\n\t Enter Size of Array:");  size = scan.nextInt();    arr = new int[size];    createArray(arr,size);  showArray(arr);  System.out.println("\n\t Max Value = " + showMax(arr));    System.out.println("\n\t Min Value = " + showMin(arr));  }    public static void createArray(int x[],int size)  {  scan = new Scanner(System.in);  System.out.print("\n\t Enter "+ size +" Array values :"); for(int i=0;i<size;i++)  {  x[i] = scan.nextInt();  }  System.out.println("\n\t Array Created.");  }    public static void showArray(int x[])  {  System.out.print("\n\t Array values are :");  for(Object obj:x)  { |

Program N0.

55

|  | }  publi  { | System.out.print(" " + obj); }  c static int showMax(int x[])  int max = x[0]; //10 for(int i=0;i<x.length;i++)  {  if(x[i]>max)  {  max = x[i];  }  } return max; |
| --- | --- | --- |
|  | } publi  { | c static int showMin(int x[])  int min = x[0]; //10 for(int i=0;i<x.length;i++)  {  if(x[i]<min)  {  min = x[i];  }  } return min; |
| } | } | |

| **Input an array of 5 elements and search the given number** |
| --- |
|  |

Program N0.

56

| **Input an array of 5 elements and count the occurrences of given number** |
| --- |
|  |

Program N0.

57

| **Input an array of Strings and store names** |
| --- |
|  |

Program N0.

58

# Array Methods



Besides the fundamental operations like getting and setting, Java provides various functions to manipulate arrays in the **Arrays** class.

The Arrays is a utility class which can be found in the java.util package. Here are some noteworthy methods it provides:

* **asList()**: returns a fixed-size list backed by an array.
* **binarySearch()**: searches for a specific value in an array. Returns the index of the element if found, or -1 if not found. Note that the array must be sorted first.
* **copyOf()**: copies a portion of the specified array to a new one.
* **copyOfRange()**: copies a specified range of an array to a new one.
* **equals()**: compares two arrays to determine if they are equal or not.
* **fill()**: fills same values to all or some elements in an array.
* **sort()**: sorts an array into ascending order.
* And other methods you can find in the Arrays class Javadoc.

In addition, the System.arraycopy() is an efficient method for copying elements from one array to another. Remember using this method instead of writing your own procedure because this method is very efficient.

## Copying Arrays

The **System** class has an **arraycopy** method that you can use to efficiently copy data from one array into another:

public static void arraycopy(Object src, int srcPos,

Object dest, int destPos, int length)

The two **Object** arguments specify the array to copy *from* and the array to copy *to*. The three **int** arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, [**ArrayCopyDemo**,](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ArrayCopyDemo.java) declares an array of **char** elements, spelling the word "decaffeinated." It uses the **System.arraycopy** method to copy a subsequence of array components into a second array:

class ArrayCopyDemo {

public static void main(String[] args) {

char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',

'i', 'n', 'a', 't', 'e', 'd' };

char[] copyTo = new char[7];

System.arraycopy(copyFrom, 2, copyTo, 0, 7);

System.out.println(new String(copyTo));

}

}

| **Passing Arrays to Methods:**  **Just as you can pass primitive type values to methods, you can also pass arrays to methods** |
| --- |
| import java.util.Scanner;    public class ArrayDemo59  {  static void showArray(int[] args)  {  System.out.println("\n\t Inside Method:Values are :"); for(int value:args)  {  System.out.print("\t" + value);  }  }    public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int[] arr= new int[5];    System.out.print("\n\t Enter 5 Values :"); for(int i=0; i<arr.length; i++)  {  arr[i] = scan.nextInt();  }  showArray(arr); |

**Progra m N0.**

**59**

| } | } |
| --- | --- |

| **Returning an Array from a Method: A method may also return an array. Program to return reverse of array using method** |
| --- |
| import java.util.Scanner;    public class ArrayDemo60  {  static int[] reverseArr(int[] list)  {  int[] result = new int[list.length]; int j = result.length-1; for(int i=0; i<list.length; i++)  {  result[j] = list[i];  j--;  }  return result;  }  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int[] arr= new int[5];    System.out.print("\n\t Enter 5 Values :"); for(int i=0; i<arr.length; i++)  {  arr[i] = scan.nextInt();  }  arr = reverseArr(arr);  System.out.println("\n\t Reverse :");  for(int value : arr)  {  System.out.print("\t" + value);  }  }  } |

**Progra m N0.**

**60**

# Array of Objects



Arrays are capable of storing objects also. For example, we can create an array of Strings which is a reference type variable.

However, using a String as a reference type to illustrate the concept of array of objects isn't too appropriate due to the immutability of String objects.

Therefore, for this purpose, we will use a class Student containing a single instance variable marks

| class Student  {  int roll; String name;  void getStud(int r, String n)  {  roll = r; name = n;  }  void showStud()  {  System.out.print("\n\t" + roll + "\t" + name);  }  } |
| --- |

0 1 2 

Hold Each element

Reference is Ref Var of of three Student

Type.. Holds ref. to

**Student[] stud;**

* The statement creates the array which can hold references to Student objects. It doesn't create the Student objects themselves.
* They have to be created separately using the constructor of the Student class. The student Array contains three memory spaces in which the address of three Student objects may be stored.

stud = new Student[3]; Stud[0] = new Student();

Stud[1] = new Student();

Stud[2] = new Student();

| **Array of Objects** |
| --- |
| class Student  {  int roll; String name;    void getStud(int r, String n)  {  roll = r; name = n;  }  void showStud()  {  System.out.print("\n\t" + roll + "\t" + name);  }  }  public class ArrayDemo61  {  public static void main(String[] args)  {  Student[] stud; stud = new Student[3];    /\* stud[0] = new Student(); stud[1] = new Student(); |

**Progra m N0.**

**61**

|  |  | stud[2] = new Student();\*/  for(int i=0; i<stud.length; i++)  {  stud[i] = new Student();  }  stud[0].getStud(101, "Bhartesh"); stud[1].getStud(102, "Payal"); stud[2].getStud(103, "Pankaj");  for(int i=0; i<stud.length; i++)  {  stud[i].showStud(); } |
| --- | --- | --- |
| } | } |  |

| **Array of Objects Employee Class** |
| --- |
| import java.util.Scanner; class Employee  {  int empid; String name; double bsal;    void getEmp(int id, String n, double sal)  {  empid = id; name = n;  bsal = sal;  }  void processSal()  {  double da = bsal\*20/100; double hra = bsal\*40/100;  double gsal = bsal+da+hra;    System.out.println("\n\t " + empid + "\t" + name +  "\t" + gsal);  }  }  public class ArrayDemo62  {  public static void main(String[] args)  {  int choice;  Scanner scan = new Scanner(System.in);  Employee[] emparr = new Employee[5]; int cnt = 0;    for(int i=0; i<emparr.length; i++)  {  emparr[i] = new Employee(); |

**Progra m N0.**

## **62**

| }    do  {  System.out.println("\n\t 1. Add Employee \n\t 2.  Process Sal \n\t 3. Quit ");  System.out.print("\n\t Enter your choice :");  choice = scan.nextInt();    switch(choice)  {  case 1:  System.out.print("\n\t Enter id:"); int id = scan.nextInt();    System.out.print("\n\t Enter Name :");  String name = scan.next();    System.out.print("\n\t Enter Bsal :"); double sal = scan.nextDouble();    emparr[cnt].getEmp(id, name, sal);  cnt++; break; case 2:  System.out.println("\n\n\t Salary  Processed");  for(int i=0; i<cnt;i++)  {  emparr[i].processSal();  }  break;    case 3: break;    default:  System.out.println("Invalid Input");  }//switch ends  }while(choice!=3);  }  } |
| --- |

| **Array of Objects Bank Account Class** |
| --- |
| import java.util.Scanner; class BankAcc  {  static int number; int accno; String name; float balance;  Scanner scan = new Scanner(System.in);    void openAccount() |

**Progra m N0.**

## **63**

| {  accno = generateAccNumber();  System.out.println("\n\t Account Number = " + accno);  System.out.print("\n\t Enter your Name :");  name = scan.next();    System.out.print("\n\t Enter Initial Amount :");  balance = scan.nextFloat();  }    private int generateAccNumber()  {  return ++number;  }    public void viewAccount()  {  System.out.println("\n\t Account Number = " + accno);  System.out.println("\n\t Account Holder Name = " + name); System.out.println("\n\t Total Available Balance = Rs. " + balance);  }    }  class Java62  {  static int cnt;  static BankAcc acc[] = new BankAcc[10];    public static void main(String []args)  {  int choice,accnumber,pos; float amt;  Scanner scan = new Scanner(System.in);    do  {  System.out.println("\n\n \*\* Menu \*\*");  System.out.println("\n 1. Open Acc \n 2. View Acc \n 3.  Deposite");  System.out.println(" 4. Withdraw \n 5. Exit");    System.out.print("\n\t Enter Choice :");  choice = scan.nextInt();    switch(choice)  {  case 1:  acc[cnt] = new BankAcc();  acc[cnt].openAccount(); cnt++;  break; |
| --- |

| case 2:  System.out.print("\n\t Enter Account Number to  Search    :");  accnumber = scan.nextInt();    pos = searchAcc(accnumber);  if(pos>=0)  {  acc[pos].viewAccount();  }  else  {  System.out.println("\n\t Accuont not found.");  }  break;    case 3:  System.out.print("\n\t Enter Account Number to  Search :");  accnumber = scan.nextInt();    pos = searchAcc(accnumber);  if(pos>=0)  {  System.out.print("\n\t Enter Amount :");  amt = scan.nextFloat();    acc[pos].balance = acc[pos].balance + amt; System.out.println("\n\t Amuont Rs. " + amt  + " is Deposited Successfully.");  }  else  {  System.out.println("\n\t Accuont not found.");  } break; case 4: break; case 5: break;    default:  System.out.println("\n\t Invalid Case.");  }  }  while(choice!=5);  }  // 1 2 3 4 5  static int searchAcc(int accno)  {  int flag=-1; | | | | |
| --- | --- | --- | --- | --- |
|  |  |  | for(i  { | nt i=0;i<cnt;i++)  if(accno==acc[i].accno)  {  flag = i;  } |
|  |  |  | } if(flag>=0)  return flag;  else  return -1; | |
| } | } |  |  | |

# Multidimensional Arrays



In some cases, we may want to divide the list in delimited sections. For example, if we create a list of names, we may want one part of the list to include family members and another part of the list to include friends.

Instead of creating a second list, we can add a second dimension to the list. In other words, we would create a list of a list, or one list inside of another list, although the list is still made of items with common characteristics

A multidimensional array is a series of arrays so that each array contains its own sub-array(s)

## Two Dimensional Arrays

The most basic multidimensional array is made of two dimensions. This is referred to as two-dimensional. To create a two-dimensional array, inside of one pair of square brackets, use two pairs (of square brackets)

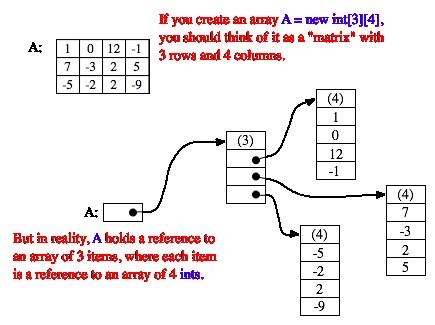
DataType[][] VariableName;

Two-dimensional arrays are used whenever the model data is best represented with rows and columns, or has two varying aspects (eg, gender and age, weight and height, ...)

Visualizing two-dimensional arrays

2-dimensional arrays are usually represented with a row-column "spreadsheet" style. Assume we have an array arr with two rows and four columns int[][] a = new int[2][4]; // Two rows and four columns

| **a[0][0]** | **a[0][1]** | **a[0][2]** | **a[0][3]** |
| --- | --- | --- | --- |
| **a[1][0]** | **a[1][1]** | **a[1][2]** | **a[1][3]** |

String[][] members = new String[2][4]; 

In above declaration, the members variable contains two lists. Each of the two lists contains 4 elements. This means that the first list contains 4 elements and the second list contains 4 elements. Therefore, the whole list is made of 8 elements (2 \* 4 = 8). Because the variable is declared as a String, each of the 8 items must be a string

## Initialize values

* You can assign initial values to an array in a manner very similar to one-dimensional arrays, but with an extra level of braces.
* The dimension sizes are computed by the compiler from the number of values. Example

int[][] board = new int[][]{ {1,0,0}, {0,1,0}, {1,2,1} };

| **Simple Program Multidimensional Array** |
| --- |
| public class ArrayDemo64  { public static void main(String[] args)  {  int[][] arr; arr = new int[2][4];    System.out.println("\n\t length of arr = " + arr.length);  System.out.println("\n\t length of arr[0] = " + arr[0].length);  System.out.println("\n\t length of arr[1] = " + arr[1].length);    arr[0][0] = 10; arr[0][1] = 20; arr[0][2] = 30; arr[0][3] = 40;    arr[1][0] = 50; arr[1][1] = 60; arr[1][2] = 70; arr[1][3] = 80;    System.out.println("\n\t value of arr[0][0] = " + arr[0][0]);  System.out.println("\n\t Values using for loop :");    for(int i=0; i<arr.length;i++)  {  for(int j=0; j<arr[i].length; j++)  { |

**Progra m N0.**

**64**

|  |  | } | System.out.print(" " + arr[i][j]); }  System.out.println("\n"); |
| --- | --- | --- | --- |
| } | } |  |  |

| **Input and Display roll no. and marks of 3 subjects** |
| --- |
| import java.util.Scanner;  public class ArrayDemo65  { public static void main(String[] args)  {  Scanner scan = new Scanner(System.in); int[][] stud = new int[5][4];  int r;  System.out.println("\n\t Enter Records :"); for(int i=0; i<stud.length; i++)  {  r = i+1;  System.out.print("\n\t Record No. : " + r); System.out.print("\n\t Enter Roll No. :"); stud[i][0] = scan.nextInt();    System.out.print("\n\t Enter Sub1 :"); stud[i][1] = scan.nextInt();    System.out.print("\n\t Enter Sub2 :"); stud[i][2] = scan.nextInt();    System.out.print("\n\t Enter Sub3 :"); stud[i][3] = scan.nextInt();  }    System.out.println("\n\tRollNO\tSub1\tSub3\tSub4");  System.out.println("\n\t---------------------\n\t");    for(int i=0; i<stud.length; i++)  {  for(int j=0; j<stud[i].length; j++)  {  System.out.print("\t" + stud[i][j]);  }  System.out.println();  }  }  } |

**Progra m N0.**

**65**

## Assignment



| **Display the total number of accidents each year** |
| --- |
|  |

Program N0.

66

# Classes and Objects



* Java is an object-oriented programming language therefore the underlying structure of all java programs is classes
* Everything to be represented in a java program must be encapsulated in a class that defines the state and behavior of objects
* A class is a user defined data type. Once the class type has been defined, we can create “variables” of that type. These “variables” are called instances or objects of classes
* An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system. ▪ An object has three characteristics: o **state:** represents data (value) of an object. o **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc. o **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely
* For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.
* **Object is an instance of a class** o Class is a template or blueprint from which objects are created. So object is the instance(result) of a class
* A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

| Syntax to declare a class: class <class\_name>  {  datamember; method();  } |
| --- |

* A class in java can contain:
  + **Data member** o **Method** o **Constructor** o **Block**
  + **Class and interface**
* Classes create objects and objects use methods.
* Classes provide a convenient method for packing together a group of logically related data items and functions.
* In java the data items are called fields and the functions are called methods.

* **Object -** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors wagging, barking, eating. An object is an instance of a class.
* **Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.



# Encapsulation



Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

| **Class Student** |
| --- |
| class Student  {  private int roll; //fields/data members/instance memberes private String name;    public void createStudent(int r, String n)  {  roll = r;  name = n;  }  public void showStudent()  {  System.out.println("\n\t Roll No. :" + roll);  System.out.println("\n\t Name : " + name);  }  }  public class ClassDemo1\_67  {  public static void main(String[] args)  {  Student stud1 = new Student(); stud1.createStudent(101, "Kavita");  stud1.showStudent();  }  } |

Program N0.

67

## Methods

* A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console.

Creating Method

* Method definition consists of a method header and a method body modifier returnType nameOfMethod (Parameter List)

{

// method body

}

* **Modifier**: It defines the access type of the method and it is optional to use.
* **Return Type**: Method may return a value.
* **Name of method**: This is the method name. The method signature consists of the method name and the parameter list.
* **Parameter** **List**: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
* **Method** **body**: The method body defines what the method does with statements.

| **Class Employee** |
| --- |
| class Employee  {  private int empId; private String empName; private double sal;    public void getEmp(int id, String ename, double amt)  {  empId = id; empName = ename; sal = amt;  }  public void showEmp()  {  System.out.println("\n\t Emp Id = " + empId);  System.out.println("\n\t Emp Name = " + empName);  System.out.println("\n\t Salary = " + sal);  }  }  public class ClassEmployee68  {  public static void main(String[] args)  {  Employee emp1 = new Employee();  emp1.getEmp(); emp1.showEmp();    }  } |

Program N0.

68

## Passing Parameters by Value

* Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter
* While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference

| **Class Product** |
| --- |
| class Product  {  private int pid;  private double price, qty, bamt;    public void getProduct(int id, double pr, double q)  {  pid = id; price = pr;  qty = q;  } |

Program N0. 69

| public void calcBamt()  {  bamt = price\*qty;  System.out.println("Product Id = " + pid);  System.out.println("Price = " + price);  System.out.println("Quantity = " + qty);  System.out.println("Bill Amount = " + bamt);  }    }  public class ProductDemo  {  public static void main(String[] args)  {  Product toy1 = new Product();  toy1.getProduct(101, 350.50, 2.00);  toy1.calcBamt();  }  } |
| --- |

| **Class Rectangle** |
| --- |
| class Rectangle  {  public int length, breadth, height;    public void getData(int l, int b, int h)  {  length = l; breadth = b;  height = h;  }    public int getArea()  {  int area;  area = length\*breadth;  return area;  }    public int getVol()  {  return (length\*breadth\*height);  }  }  public class ClassRectangle70  {  public static void main(String[] args)  {  Rectangle rect1 = new Rectangle();  rect1.getData(10,3,2);    int area;  area = rect1.getArea(); |

Program N0.

70

System.out.println("\n\t Area of Rect1 = " + area); System.out.println("\n\t Vol of Rect1 = " + rect1.getVol());

}

}

## assignments

| **Define a class BOOK with the following specifications** |
| --- |
| Private members of the class BOOK are  BOOK NO : integer type  BOOKTITLE : 20 characters  PRICE : float (price per copy)  TOTAL\_COST() : A function to calculate the total cost for N number of copies where N is passed to the function as argument    Public members of the class BOOK are  INPUT() : function to read BOOK\_NO. BOOKTITLE, PRICE  PURCHASE() : function to ask the user to input the number of copies to be purchased. It invokes TOTAL\_COST() and prints the total cost to be paid by the user. Note : You are also required to give detailed function definitions |

Program N0.

71

| **Define a class in Java with following description:** |
| --- |
| Private Members  A data member Flight number of type integer  A data member Destination of type string  A data member Distance of type float  A data member Fuel of type float  A member function CALFUEL() to calculate the value of Fuel as per the following criteria  Distance Fuel  <=1000 500  more than 1000 and <=2000 1100 more than 2000 2200  Public Members  A function FEEDINFO() to allow user to enter values for Flight Number, Destination, Distance & call function CALFUEL() to calculate the quantity of Fuel  A function SHOWINFO() to allow user to view the content of all the data members |

Program N0.

72

| **Define a class in Java with following description:** |
| --- |
| Private members:  bcode : 4 digits code number bname : String  innings, notout, runs : integer type  batavg : it is calculated according to the formula  : batavg =runs/(innings-notout)  calcavg() : Function to compute batavg Public members:  readdata() : Function to accept value from bcode, name, innings, notout |

Program N0. 73

and invoke the function calcavg()

displaydata() : Function to display the data members on the screen

What is Overloading?



Overloading refers to the process of having multiple methods or constructors using a common name but with different argument list. The methods can be in the same class or in both super class and sub class.

* When a class has two or more methods by same name but different parameters, it is known as method overloading
* Method overloading allows to define same name for two or more methods, providing that the number of parameters or the data type of parameters are different.
* Method overloading is one way to realize polymorphism, which means to allow one interface to be used with multiple methods.
* To declare more than one methods, all methods must be declared with different argument list.
* The methods would perform different operations depending on the argument list in the method call.
* The correct method to be invoked is determined by checking the number and type of arguments.

Example #1:

public class Dog { private String name;

public Dog() {

[this.name](http://this.name/) = "Puppy";

} public Dog(String name) {  [this.name](http://this.name/) = name;

} }

Here, the class Dog has two overloaded constructors: the first one with no argument; and the second one with a String argument (for name). Then we have two ways for creating Dog objects:

Dog dog1 = new Dog(); // name is 'Puppy' Dog dog2 = new Dog("Rex"); // name is 'Rex'

Example #2:

public class Cat {

public void catches(Mouse mouse) {

System.out.print("Catching a mouse...");

}

public void catches(Fish fish) {

System.out.print("Catching a fish...");

}

}

Here, the Cat class has two overloaded methods: one for catching a mouse; and another for catching a fish. Both has same name (catches) but argument type is different (Mouse and Fish). So we can write:

Cat myCat = new Cat(); myCat.catches(new Mouse()); myCat.catches(new Fish());

Example #3:

public class Lion extends Cat {

public void catches(Buffalo buffalo) { System.out.print("Catching a buffalo...");

}

}

As you can see, this Lion class is a sub type of the Cat class. The Lion class overloads the catches() method in order to catch a buffalo. In this case, the method in the super class is called overloaded method, and the method in the sub class is called overloading method. We can write:

Lion lion = new Lion(); lion.catches(new Mouse()); lion.catches(new Buffalo());

Why is Overloading?

At low level, overloading is for re-using method names and constructor names depending on needs. For example, you may want to create a Dog with default name ‘Puppy’, or sometimes you want to create a Dog with your own name; Sometimes you may want your cat to catch a mouse, but other times he can catch a fish, etc.

At high level, overloading is for extending functionalities but keeping the flexibility of code.

Overloading Rules

Compared to overriding, overloading is simpler, thus the rules are more relaxed.

Here are the rules with regard to overloaded methods:

* Overloaded methods must have different argument list.
* Overloaded methods may have different return type (the argument list is different as well).
* Overloaded methods may have different access modifiers.
* Overloaded methods may throw different exceptions.

| **Arithmetic Operations** |
| --- |
| public class MethodOverloadDemo\_74  {  static void arithAdd(int a, int b)  {  System.out.println("\n\t This is Method No. 1");  System.out.println("\n\t Num1 = " + a);  System.out.println("\n\t Num2 = " + b); int sum = a + b;  System.out.println("\n\t Addition = " + sum);  }  static void arithAdd(int a, double b)  {  System.out.println("\n\t This is Method No. 2");  System.out.println("\n\t Num1 = " + a);  System.out.println("\n\t Num2 = " + b); double sum = a + b;  System.out.println("\n\t Addition = " + sum);  } |

Program N0. 74

|  | stati  { | c double arithAdd(double a, int b)  System.out.println("\n\t This is Method No. 3");  System.out.println("\n\t Num1 = " + a); System.out.println("\n\t Num2 = " + b); double sum = a+b; return sum; |
| --- | --- | --- |
| } | }  public static void main(String[] args)  {  arithAdd(10,20);  } | |

| **Class BankAccount** |
| --- |
| class BankAccount  {  private int accId; private String custName; private double bal;    public void openAccount()  {  System.out.println("\n\t Bank Acc Opened");  }  public void openAccount(int id, String name)  {  System.out.println("\n\t Bank Account opened with Following  Details :");  accId = id; custName = name;  bal = 0.00;  System.out.println("\n\t Account Id = " + accId);  System.out.println("\n\t Cust Name = " + custName);  System.out.println("\n\t Balance = " + bal);  }  public void openAccount(int id, String name, double iniamt)  {  System.out.println("\n\t Bank Account opened with Initial  Amount:");  accId = id; custName = name; bal = iniamt;  System.out.println("\n\t Account Id = " + accId);  System.out.println("\n\t Cust Name = " + custName);  System.out.println("\n\t Balance = " + bal);  }  }  public class BankAccountDemo\_75  {  public static void main(String[] args)  {  BankAccount SB1 = new BankAccount(); |

Program N0.

75

|  |  | SB1.openAccount();  BankAccount SB2 = new BankAccount(); SB2.openAccount(101, "Bhartesh");  BankAccount SB3 = new BankAccount();  SB3.openAccount(101, "Payal", 1000.00); |
| --- | --- | --- |
| } | } |  |

# The Constructors



* A class constructor is a special member function of a class that is executed whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
* A constructor function has exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.
* Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.

| Syntax  class <ClassName>  {  …datamembers/fields;  public <ClassName>(argument list)  {  …statements;  } |
| --- |
| **Class Demo** |
| class Demo  {  int x, y;  public Demo() //Constructor Method  {  x = 10; y = 20;  }  void show()  {  System.out.println("\n\t x = " + x);  System.out.println("\n\t y = " + y);  }  }  public class ConstructorDemo\_76  {  public static void main(String[] args)  {  Demo obj1 = new Demo(); obj1.show();  }  } |

} Program N0.

76

## Parameterized Constructor

* Constructors may include parameters of various types. When a constructor is parameterized the object declaration statement must pass the initial values to the parameters

| **Class Rectangle - Parameterized Constructor** |
| --- |
| import java.util.Scanner;    class Rectangle  {  public int length, breadth;    public Rectangle(int l, int b)  {  length = l;  breadth = b;  }    public void calcArea()  {  int area = length \* breadth;  System.out.println("\n\t Length = " + length);  System.out.println("\n\t Breadth = " + breadth);  System.out.println("\n\t Area = " + area);  }  }  public class RectangleDemo  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int l, b;    System.out.print("\n\t Enter Length :");  l = scan.nextInt();    System.out.print("\n\t Enter Breadth :");  b = scan.nextInt();    Rectangle rect1 = new Rectangle(l, b);  rect1.calcArea();  }  } |

* When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition Program N0.

77

## Overloaded Constructors

▪ Like methods, constructors can also be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists

| **Class Bank Account** |
| --- |
| class BankAccount  {  int id;  String name; double bal;    public BankAccount() |

Program N0. 78

| {  System.out.println("Bank Acc Opened");  }    public BankAccount(int i, String n)  {  id = i; name = n;  bal = 0.00;  System.out.println("\n\t Bank Acc Opened with  Following Details :");  System.out.println("\n\t id = " + id);  System.out.println("\n\t Name = " + name);  System.out.println("\n\t Balance = " + bal);  }  public BankAccount(int i, String n, double iniamt)  {  id = i; name = n; bal = iniamt;  System.out.println("\n\t Bank Acc Opened with  Following Details :");  System.out.println("\n\t id = " + id);  System.out.println("\n\t Name = " + name);  System.out.println("\n\t Balance = " + bal);  }  }  public class BankAccountDemo\_78  {  public static void main(String[] args)  {  BankAccount SB1 = new BankAccount(101, "asdfjdsf", 1000.00);  }  } |
| --- |

## this() constructor

* “**this**” keyword in Java is a special keyword which can be used to represent current object or instance of any class in Java
* “**this**()” statement is used to call constructor of same class in Java (used to call overloaded constructor)

| **this() statement with Constructor Class Goods** |
| --- |
| class Goods  {  int id;  String name; String description;  public Goods()  { |

* It is called Explicit Constructor Invocation. If a class has two overloaded constructor one without argument and another with argument. Then this keyword can be used to call Constructor with argument from Constructor without argument Program N0. 79

| System.out.println("\n\t Goods Created");  }  public Goods(int i, String n)  {  this(); id = i;  name = n;  System.out.println("\n\t id = " + id);  System.out.println("\n\t Name = " + name);  }  public Goods(int i, String n, String des)  {  this(i, n); description = des;  System.out.println("\n\t Description = " + description);  }    }  public class ThisConstructorDemo  {  public static void main(String[] args)  {  Goods g1 = new Goods(101, "Mouse", "Optical Mouse");  }  } |
| --- |

Note

* this keyword can only be the first statement in Constructor
* A constructor can have either this or super keyword but not both ▪ this is a final variable in Java and you cannot assign value to this

“this” to refer current class instance variable

* If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity

| **this keyword to refer current class instance variable** |
| --- |
| class Goods  {  int id;  String name;  String description;    public Goods()  {  System.out.println("\n\t Goods Created");  }  public Goods(int id, String name)  {  this();  this.id = id; |

Program N0. 80

| this.name = name;  System.out.println("\n\t id = " + id);  System.out.println("\n\t Name = " + name);  }  public Goods(int id, String n, String des)  {  this(id, n); description = des;  System.out.println("\n\t Description = " + description);  }  }  public class ThisInstanceVar\_80  {  public static void main(String[] args)  {    }  } |
| --- |

“this” passed as an argument in the method

* The **this** keyword can also be passed as an argument in the method. It is mainly used in the event handling

| **“this” passed as an argument in the method** |
| --- |
|  |

Program N0.

81

## This keyword with Method

▪ this keyword can also be used inside methods to call another method from same class

| **“this” keyword with method** |
| --- |
| class Demo  {  private void method1()  {  System.out.println("\n\t Method is Invoked");  }    public void method2()  {  this.method1();  }    }  public class ThisMethod\_82  {  public static void main(String[] args)  {  Demo d = new Demo(); d.method2();  } |

Program N0. 82

}

# String

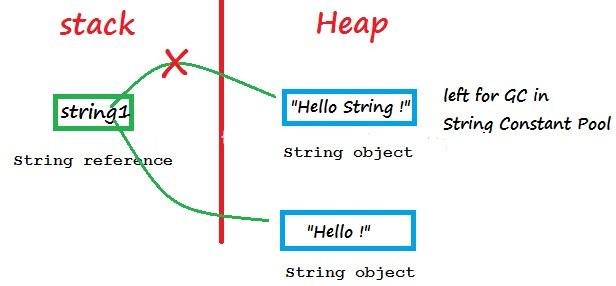


A Java string is a series of characters gathered together, like the word "Hello", or the phrase "practice makes perfect". Create a string in the code by writing its chars out between double quotes.

* Strings are a sequence of characters
* In the Java, strings are objects
* The Java platform provides the String class to create and manipulate strings ▪ Once a String object is created it cannot be changed. Stings are Immutable.
* There are various ways of creating String object
  + Direct method
  + Using String class constructors
  + Using toString() method defined in java.lang.Object

## Creating Strings

The String class has eleven constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.



| **String Demo** |
| --- |
| public class StringDemo\_83  {  public static void main(String[] args) |

Program N0. 83

|  | { | String str1 = "Hello World";  System.out.println("\n\t str1 = " + str1);  str1 = "Hello!";  System.out.println("\n\t str1 = " + str1); |
| --- | --- | --- |
| } | } |  |

Note:

The String class is immutable; so that once it is created a String object cannot be changed.

If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes

### String Length

* Methods used to obtain information about an object are known as accessor methods.
* One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

| **length() method** |
| --- |
| public class StringDemo\_84  {  public static void main(String[] args)  {  String str1 = "Hello World";  System.out.println("\n\t str1 = " + str1);  int len = str1.length();  System.out.println("\n\t Length of str1 = " + len);  }  } |

Program N0.

84

### Concatenating Strings

* The String class includes a method for concatenating two strings:

string1.concat(string2);

* This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals.

| **Concat() method** |
| --- |
| public class StringDemo\_85  {  public static void main(String[] args)    String fname= "Vishal"; String lname= "Shah"; fname = fname.concat(" ");  String fullName = fname.concat(lname);  System.out.println("\n\t fname = " + fname);  System.out.println("\n\t lname = " + lname);  System.out.println("\n\t Fullname = " + fullName); |

Program N0. 85

| } | } |
| --- | --- |

## String class Methods



charAt()

* public o Returns the character at the specified index. An index ranges from 0 to length() - 1. o The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing

char ch;

ch = “abc”.charAt(1); // ch = “b”

getChars()

* Copies characters from this string into the destination character array
* public void **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) o **srcBegin** - index of the first character in the string to copy. o **srcEnd** - index after the last character in the string to copy. o **dst** - the destination array. o **dstBegin** - the start offset in the destination array

| **charAt() getChars()** |
| --- |
| public class StringDemo\_86  {  public static void main(String[] args)  {  String src = "This is Source";  char[] chararr = {'a','b',  'c','d','e','f','g','h','i','j','k'};    src.getChars(0,3,chararr,5); String dest = new String(chararr);    System.out.println("src = " + src);  System.out.println("des = " + dest);    }  } |

Program N0.

86

compareTo()

* Compares two strings lexicographically o The result is a negative integer if this String object lexicographically precedes the argument string.
  + The result is a positive integer if this String object lexicographically follows the argument string o The result is zero if the strings are equal
  + compareTo returns 0 exactly when the equals(Object) method would return true
* public int compareTo(String anotherString)
* public int compareToIgnoreCase(String str)

equals()

public boolean equals(Object anObject)

* Compares the invoking string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as the invoking object.

equalsIgnoreCase()

* Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

public boolean equalsIgnoreCase(String anotherString)

| **compareTo() Equals()**  **equalsIgnoreCase()** |
| --- |
| public class StringDemo\_87  {  public static void main(String[] args)  {  String str1 = "Abc";  String str2 = "abc";    int result = str1.compareTo(str2);    System.out.println("str1 = " + str1);  System.out.println("str2 = " + str2);  System.out.println("Result = " + result);    if(str1.equalsIgnoreCase(str2))  System.out.println("String are Same");  else  System.out.println("Strings are not same");    }  } |

Program N0.

87

startsWith()

public boolean startsWith(String prefix)

* Tests if this string starts with the specified prefix.

endsWith()

public boolean endsWith(String suffix)

* Tests if this string ends with the specified suffix.

| **startsWith() endsWith()** |
| --- |
| public class StringDemo\_88  { public static void main(String[] args)  {  String mob = "985081h100";    if(mob.length()==10)  { |

Program N0. 88

| if(mob.startsWith("7") || mob.startsWith("8") || mob.startsWith("9"))  {  System.out.println("Valid Mobile");  } else  {  System.out.println("Invalid");  }  }  else  {  System.out.println("Invalid Mobile No.");  }  }  } |
| --- |

indexOf

* Searches for the first occurrence of a character or substring. Returns -1 if the character does not occur

public int indexOf(int ch)

* Returns the index within this string of the first occurrence of the specified character.

public int indexOf(String str)

* Returns the index within this string of the first occurrence of the specified substring.

public int indexOf(int ch, int fromIndex)

* Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

public int indexOf(String str, int fromIndex)

* Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

lastIndexOf()

* Searches for the last occurrence of a character or substring. The methods are similar to indexOf().

| **indexOf()** |
| --- |
|  |

Program N0.

89

replace()

public String replace(char oldChar, char newChar)

* Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

trim()

public String trim()

* Returns a copy of the string, with leading and trailing whitespace omitted.
* **toLowerCase():** Converts all of the characters in a String to lower case.
* **toUpperCase():** Converts all of the characters in this String to upper case.

| **replace() Trim()** |
| --- |

Program N0. 90

| **toLowerCase() toUpperCase()** |
| --- |
|  |

# StringBuilder class



* The StringBuilder class is used to create mutable (modifiable) string.
* The StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK1.5

* StringBuilder() creates an empty string Builder with the initial capacity of 16 ▪ StringBuilder(String str): creates a string Builder with the specified string.
* StringBuilder(int length): creates an empty string Builder with the specified capacity as length

Commonly used methods of StringBuilder class

public StringBuilder append(String s)

* Used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

public StringBuilder insert(int offset, String s)

* Used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

public StringBuilder replace(int startIndex, int endIndex, String str) ▪ Used to replace the string from specified startIndex and endIndex.

public StringBuilder delete(int startIndex, int endIndex)

* Used to delete the string from specified startIndex and endIndex.

public StringBuilder reverse() ▪ Used to reverse the string.

public int capacity()

* Used to return the current capacity.

public char charAt(int index)

* used to return the character at the specified position.

public int length()

* used to return the length of the string i.e. total number of characters.

public String substring(int beginIndex)

* used to return the substring from the specified beginIndex.

public String substring(int beginIndex, int endIndex)

* used to return the substring from the specified beginIndex and endIndex

| **StringBuilder Class methods : Append() Insert() Replace() Delete()** |
| --- |
| public class StringBuilderDemo\_91  {  public static void main(String[] args)  {  StringBuilder sb = new StringBuilder("abcd");  System.out.println("\n\t sb = " + sb); |

Program N0. 91

|  |  | sb = sb.append("newstring");  System.out.println("\n\t After Append: sb = "+ sb);  sb = sb.insert(1, "www");  System.out.println("\n\t After insert : sb = " + sb);  sb = sb.replace(1, 4, "http");  System.out.println("\n\t After REplace :sb = " + sb);  sb = sb.delete(1,5);  System.out.println("\n\t After Delete : sb = " + sb); |
| --- | --- | --- |
| } | } |  |

| **StringBuilder Methods : Reverse() Capacity() Substring()** |
| --- |
| public class StringBuilderDemo\_92  {  public static void main(String[] args)  {  StringBuilder sb = new StringBuilder("abcdefghijk");  System.out.println("\n\t sb = " + sb);    sb = sb.reverse();  System.out.println("\n\t Reverse sb = " + sb);  System.out.println("\n\t Capacity = " + sb.capacity());    System.out.println("\n\t Substring : " + sb.substring(0,4));  }  } |

Program N0.

92

# StringBuffer Class



* The Java StringBuffer class is just like StringBuilder. The primary difference is that StringBuffer is synchronized. In other words, multiple threads can safely process StringBuffer objects.
* StringBuffer is slower than StringBuilder. You should use the StringBuffer class for string objects that will be changed by multiple threads of execution.

# Immutable class



* There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable.
* We can also create immutable class by creating final class that have final data members

| **Immutable class : Employee** |
| --- |
| final class Employee  {  final String panCardNo; |

Program N0. 93

| public Employee(String no)  {  panCardNo = no;  }    public String getEmpPanNo()  {  return panCardNo;  }  }  public class ImmutableClassDemo\_93  {  public static void main(String[] args)  {  Employee emp1 = new Employee("BLZPS8025A");  System.out.println("Pan = " + emp1.getEmpPanNo());  }  } |
| --- |

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There are no setter methods i.e. we have no option to change the value of the instance variable. toString() method



* The toString() method returns the string representation of the object
* If you print any object, java compiler internally invokes the toString() method on the object.
* So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation
* By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code

| **Override toString() method** |
| --- |
| class Student  {  int roll; String name;  int marks;    public Student()  {  roll = 101; name = "Bhartesh";  marks = 45;  }    public String toString()  {  return roll + "\t" + name + "\t" + marks;  }  } |

Program N0. 94

public class ToStringMethod\_94

{

public static void main(String[] args)

{

Student s1 = new Student();

System.out.println("\n\t " + s1);

}

}

# Java Regex



* The Java Regex or Regular Expression is an API to define pattern for searching or manipulating strings ▪ It is used to define constraint on strings such as password and email validation.

It provides following classes and interface for regular expressions. The Matcher and Pattern classes are widely used in java regular expression.

* MatchResult interface
* Matcher class
* Pattern class

• PatternSyntaxException class

## Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

| No. | Method | Description |
| --- | --- | --- |
| 1 | static Pattern compile(String regex) | compiles the given regex and return the instance of pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with pattern. |
| 3 | static boolean matches(String regex, CharSequence input) | It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern. |
| 4 | String[] split(CharSequence input) | splits the given input string around matches of given pattern. |
| 5 | String pattern() | returns the regex pattern. |

## Matcher class

It implements MatchResult interface. It is a regex engine i.e. used to perform match operations on a character sequence.

| No. | Method | Description |
| --- | --- | --- |
| 1 | boolean matches() | test whether the regular expression matches the pattern. |
| 2 | boolean find() | finds the next expression that matches the pattern. |
| 3 | boolean find(int start) | finds the next expression that matches the pattern from the given start number |

| **RegexDemo\_95** |
| --- |
| import java.util.Scanner; import java.util.regex.\*; |

Program N0. 95

public class RegexDemo\_95

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println(Pattern.matches("[0-9]", "23434"));

}

}

| **RegexDemo\_96** |
| --- |
| import java.util.regex.\*; public class RegexExample3  {  public static void main(String args[])  {  boolean isValidMob;  isValidMob = Pattern.matches("[0-9]{1,10}", "9ds0810100");  System.out.println(isValidMob);  }  } |

Program N0.

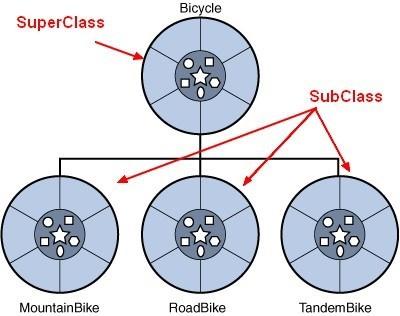
96

## Regex Character class

| No. | Character Class | Description |
| --- | --- | --- |
| 1 | [abc] | a, b, or c (simple class) |
| 2 | [^abc] | Any character except a, b, or c (negation) |
| 3 | [a-zA-Z] | a through z or A through Z, inclusive (range) |
| 4 | [0-9] | 0 through 9 |

# Inheritance



* Inheritance is the ability of a class to inherit from data and behaviors another class to provide the reusability. Inheritance is one of the key features of Object Oriented Programming.
* When a Class extends another class, it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language 
* A class that is derived from another class is called a subclass (also a *derived class*, *extended class*, or *child class*).
* The class from which the subclass is derived is called a superclass (also a *base class* or a *parent class*).
* A class can be defined as a "subclass" of another class.
  + The subclass inherits all data attributes of its superclass
  + The subclass inherits all methods of its superclass - The subclass inherits all associations of its superclass ▪ The subclass can:
  + Add new functionality
  + Use inherited functionality
  + Override inherited functionality
* Inheritance is declared using the "extends" keyword
  + If inheritance is not defined, the class extends a class called Object public class Person

{

private String name;

private Date dob;

[...]

}

public class Employee extends Person

{

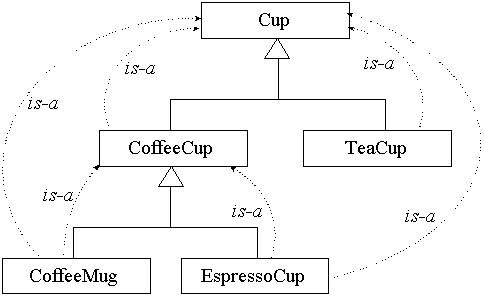
private int employeID; private int salary;

private Date startDate;

[...]

}

* Inheritance should create an **IS-A** relationship, meaning the child is a more specific version of the parent ▪ Inheritance defines **IS-A** relationship between a Super class and its Sub class.



* Syntax

**class** Superclass-Name

{

//methods and fields

}

**class** Subclass-name **extends** Superclass-name

{

//methods and fields

}

WHY IS INHERITANCE?

As you can see in the above examples, inheritance is for **reusing** **code**. When you want to extend features of a class, you can write a subclass to inherit all data and behaviors of that superclass. This saves time on writing code.

Another reason for implementing inheritance is for the purpose of **extensibility**. It’s easier to extend a class and add new features than writing a new class from scratch.

And using inheritance promotes the **maintainability** of the code. Imagine you have a superclass and 5 subclasses. When you want to update a common feature of all these classes, you just update the parent class in one place. That’s much easier than updating every single class in case there is no inheritance.

| **Inheritance Demo** |
| --- |
| class SuperDemo |

Program N0. 97

| {  int x, y;    void getxy(int a, int b)  {  x = a; y = b;  }  void showxy()  {  System.out.println("\n\t x = " + x);  System.out.println("\n\t y = " + y);  }  }  class DerivedDemo extends SuperDemo  {  int z;    void getz(int a)  {  z = a;  }  void showz()  {  System.out.println("\n\t z = " + z);  }  }  public class InheritanceDemo\_113  {  public static void main(String[] args)  {  DerivedDemo d1 = new DerivedDemo();  d1.getxy(10,20);  d1.getz(30);    d1.showxy();  d1.showz();  }  } |
| --- |

* Private members of the superclass are not inherited by the subclass
* Members that have default accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same package as the superclass
* Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.
* A subclass can extend only one superclass

| **Class Student extends Person** |
| --- |
| class Person  {  public String name;  public int age;    public void getPerson(int a, String n) |

Program N0. 98

| {  age = a;  name = n;  }  public void showPerson()  {  System.out.println("\n\t Age = " + age);  System.out.println("\n\t Name = " + name);  }  }  class Student extends Person  {  public int studid;  public int marks;    public void getStudent(int a, String n, int id, int m)  {  getPerson(a, n); studid = id;  marks = m;  }  public void showStudent()  {  System.out.println("\n\t Student id = " + studid);  showPerson();  System.out.println("\n\t Marks = " + marks);  }  }  public class InheritanceDemo\_114  {  public static void main(String[] args)  {  Student stud1 = new Student(); stud1.getStudent(25, "Amol", 101, 85);  stud1.showStudent();  }  } |
| --- |

# FORMS of Inheritance



* On the basis of class, there can be three types of inheritance
  1. Single
  2. Multilevel
  3. Hierarchical

* Multiple and Hybrid is supported through interface only

## Single Inheritance

▪ The Inheritance hierarchy where one class extends another class is called as Single Inheritance

| **Class Bicycle** |
| --- |

Program N0.

99 public class Bicycle {

// **the Bicycle class has three *fields*** public int cadence; public int gear; public int speed;

// **the Bicycle class has one *constructor*** public Bicycle(int startCadence, int startSpeed, int startGear) { gear = startGear; cadence = startCadence; speed = startSpeed;

}

// **the Bicycle class has four *methods*** public void setCadence(int newValue) { cadence = newValue;

} public void setGear(int newValue) { gear = newValue;

} public void applyBrake(int decrement) { speed -= decrement;

} public void speedUp(int increment) { speed += increment;

}

}

public class MountainBike extends Bicycle {

// **the MountainBike subclass adds one *field*** public int seatHeight;

// **the MountainBike subclass has one *constructor*** public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) { super(startCadence, startSpeed, startGear); seatHeight = startHeight;

}

// **the MountainBike subclass adds one *method*** public void setHeight(int newValue) { seatHeight = newValue;

}

}

What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

* The inherited fields can be used directly, just like any other fields.
* You can declare a field in the subclass with the same name as the one in the **superclass**, thus **hiding** it (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus **overriding**it.
* You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus **hiding** it.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword **super**.

Private Members in a Superclass

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

## Multilevel Inheritance

▪ The Inheritance Hierarchy where one derived class acts as Parent/Super class to another class is called as Multilevel Inheritance

| **Multilevel Inheritance Example** |
| --- |
| class Goods  {  public String gname;    public void getGoods(String name)  {  gname = name;  }  }  class Food extends Goods  {  int cal;  public void getFood(String name, int c)  {  getGoods(name);  cal = c;  }  }  class Jam extends Food  {  String des; |

Program N0. 100

| double price;    public void getJam(String name, int c, String d, double p)  {  getFood(name, c); des = d;  price = p;  }  public void showJam()  {  System.out.println("\n\t Name = " + gname);  System.out.println("\n\t Cal = " + cal);  System.out.println("\n\t Description = " + des);  System.out.println("\n\t Price = " + price);  }  }  public class InheritanceDemo\_116  {  public static void main(String[] args)  {  Jam j1 = new Jam();  j1.getJam("Fruit Jam", 200, "Mix Fruit Jam 200 gm",  25.30);  j1.showJam();  }  } |
| --- |

## Hierarchical Inheritance

▪ The Inheritance Hierarchy where more than one classes are derived from one Superclass

| **Hierarchical Inheritance – Bank Account Class [Assignment]** |
| --- |
|  |

Program N0.

101

# Super keyword



* The super is a reference variable that is used to refer immediate parent class object.
* Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.
* Usage of super Keyword
  + super() is used to invoke immediate parent class constructor
  + super is used to refer immediate parent class instance variable
  + super is used to invoke immediate parent class method

Calling Base Class Constructor

* In Java, constructor of base class with no argument gets automatically called in derived class constructor.

| **Calling Base Class Constructor** |
| --- |
| class BaseClass  {  public BaseClass() |

Program N0. 102

{

System.out.println("\n\t This is Base Class Default

Constructor");

}

}

class DerivedClass extends BaseClass

{

public DerivedClass()

{

System.out.println("\n\t This is Derived Class

Constructor");

}

}

public class BaseClassConstructor\_120

{

public static void main(String[] args)

{

DerivedClass d1 = new DerivedClass();

}

}

* If the superclass/baseclass has parameterized constructor then it is invoked/called using super()
* Use of super() for calling base class constructors must be the first line in derived class constructor

| **Calling Base Class parameterized Constructor** |
| --- |
| class BaseClass  {  public BaseClass()  {  System.out.println("\n\t This is Base Class Default Constructor");  }    public BaseClass(int a)  {  System.out.println("\n\t Base Paramaterized  Constructor : a = " + a);  }  }  class DerivedClass extends BaseClass  {  public DerivedClass(int a)  {  super(a);  System.out.println("\n\t This is Derived Class  Constructor");  }  }  public class BaseClassConstructor\_121  {  public static void main(String[] args)  { |

Program N0.

103

| } | DerivedClass d = new DerivedClass(10); } |
| --- | --- |

## Controlling Access to Members of a Class

* Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
* A class may be declared with the modifier public, in which case that class is visible to all classes everywhere
* If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package o (packages are named groups of related classes)
* A class may be declared with the modifier public, in which case that class is visible to all classes everywhere ▪ If a class has no modifier (the default, also known as package-private), it is visible only within its own package ▪ The private modifier specifies that the member can only be accessed in its own class.
* The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| **Public** | **Y** | **Y** | **Y** | **Y** |
| **protected** | **Y** | **Y** | **Y** | **N** |
| **no modifier** | **Y** | **Y** | **N** | **N** |
| **Private** | **Y** | **N** | **N** | **N** |

# HAS-A Relationship



* These relationships are mainly based on the usage.
* This determines whether a certain class **HAS-A** certain thing. (This relationship helps to reduce duplication of code as well as bugs)
* If a class has an entity reference (of other class), it is known as Aggregation.
* Aggregation represents HAS-A relationship.

## Example

* An Employee object contains some information such as id, name, emailId etc.
* It contains one more object named address, which contains its own information such as city, state, country, zipcode etc.

| **Has-A Relationship Example** |
| --- |
| class Address  {  public String street,area, city, pincode;  public Address(String st, String ar, String ct, String pin)  {  street = st; area = ar; city = ct;  pincode = pin;  }  public void showAddress() |

Program N0. 104

| {  System.out.println("\n\t Address = " + street + ",\n\t " + area + "," + city + ",\n\t " + pincode);  }  }  class Supplier  {  public int suppid; public String sname; public Address add;    public Supplier(int sid, String sn, Address ad)  {  suppid = sid; sname = sn; add = ad;  }  public void showSupplier()  {  System.out.println("\n\t Supplier Info = " + suppid + "\t"  + sname);  add.showAddress();  }  }  class Customer  {  public int custid; public String cname;  public Address add;    public Customer(int cid, String cn, Address ad)  {  custid = cid; cname = cn;  add = ad;  }  public void showCustomer()  {  System.out.println("\n\t Customer Info = " + custid + "\t"  + cname);  add.showAddress();  }  }  public class InheritanceDemo  {  public static void main(String[] args)  {  Address ad = new Address("338/3B, Pragati Colony", "100Ft road", "Sangli", "416415");  Supplier supp = new Supplier(101, "Devang",ad);  supp.showSupplier(); |
| --- |

Customer cust = new Customer(102, "Alfaz",new Address("100Ft Road","Vishrambag","Sangli","416415")); cust.showCustomer();

}

}

| **Has-A Relationship Example** |
| --- |
| class Operation  {  public double square(double n)  {  return n\*n;  }  }  class Circle  {  Operation op; //has-a relationship double pi = 3.14, rad;    public Circle(double r)  {  rad = r;  }    public double calcArea()  {  op = new Operation(); return pi\*op.square(rad);  }  }  public class AggragationDemo\_119  {  public static void main(String[] args)  {  Circle cir1 = new Circle(2.50);  System.out.println("\n\t Area of Circle = " + cir1.calcArea());  }  } |

Program N0.

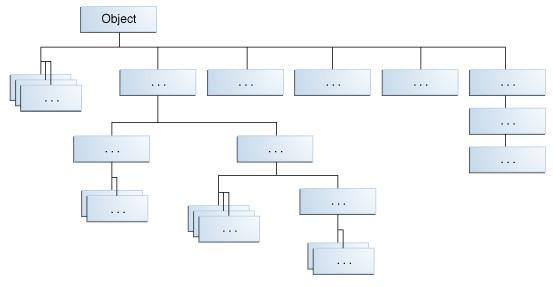
105



# The Java Platform Class Hierarchy



The [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) class, defined in the java.lang package, defines and implements behavior common to all classes— including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



* All Classes in the Java Platform are Descendants of Object
* At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

# Casting Objects



We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

public MountainBike myBike = new MountainBike(); then myBike is of type MountainBike.

MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.

The reverse is not necessarily true: a Bicycle *may be* a MountainBike, but it isn't necessarily. Similarly, an Object *may be* a Bicycle or a MountainBike, but it isn't necessarily.

**Casting** shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

Object obj = new MountainBike(); then obj is both an Object and a MountainBike (until such time as obj is assigned another object that is *not* a MountainBike). This is called *implicit casting*.

If, on the other hand, we write

MountainBike myBike = obj; we would get a compile-time error because obj is not known to the compiler to be a MountainBike. However, we can *tell* the compiler that we promise to assign a MountainBike to obj by *explicit casting:*

MountainBike myBike = (MountainBike)obj;

This cast inserts a runtime checks that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a MountainBike at runtime, an exception will be thrown.

**Note:** You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

if (obj instanceof MountainBike) {

MountainBike myBike = (MountainBike)obj; }

Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with knowledge that there will be no runtime exception thrown.

What Is Polymorphism?



Polymorphism means ‘**many forms**’. In OOP, polymorphism means a type can point to different objects at different times. In other words, the actual object to which a reference type refers, can be determined at runtime.

In Java, polymorphism is based on inheritance and overriding.

How is Polymorphism Implemented in Java?

In Java, you can implement polymorphism if you have a super class (or a super interface) with two or more sub classes.

Suppose that we have the following interface and classes:

public interface Animal { public void move();

}

public class Bird implements Animal { public void move() {

System.out.print("Flying...");

} }

public class Fish implements Animal { public void move() {

System.out.print("Swimming...");

} }

As you can see, we have Animal as the super interface, and 3 sub classes: Dog, Bird and Fish.

Because the Dog implements Animal , or Dog is an Animal, we can write:

Animal anim = new Dog();

Because Bird is an Animal, it’s legal to write:

Animal anim = new Bird();

Likewise, it’s perfect to write:

Animal anim = new Fish();

As you can see, we declare a reference variable called anim, which is of type Animal. Then we assign this reference variable to 3 different kinds of object: Dog, Bird and Fish.

You see? A reference type can take different objects (many forms). This is the simplest form of polymorphism, got it?

Now we come to a more interesting example to see the power of polymorphism.

Suppose that we have a trainer who teaches animals. We create the Trainer class as follows:

public class Trainer {

public void teach(Animal anim) {

anim.move();

}

}

Notice that the teach() method accepts any kind of Animal. Thus we can pass any objects which are sub types of the Animal type. For example:

Trainer trainer = new Trainer();

Dog dog = new Dog();

Bird bird = new Bird();

Fish fish = new Fish();

trainer.teach(dog); trainer.teach(bird); trainer.teach(fish);

Outputs:

Running…

Flying…

Swimming…

Here, as you can see, the teach() method can accept ‘many forms’ of Animal: Dog, Bird, Fish,… as long as they are sub types of the Animal interface.

In the teach() method, the move() method is invoked on the Animal reference. And depending on the actual object type, the appropriate overriding method is called. Thus we see the outputs:

*Running…* (from the Dog object).

*Flying…* (from the Dog object).

*Running…* (from the Dog object).

Why is Polymorphism?

Polymorphism is a robust feature of OOP. It increases the reusability, flexibility and extensibility of code. Take the above example for instance:

* Reusability: the teach() method can be re-used for different kinds of objects as long as they are sub types of the Animal interface.
* Flexibility: the actual object can be determined at runtime which allows the code run more flexibly.
* Extensibility: when we want to add a new kind of Animal, e.g. Snake, we just pass an object of Snake into the teach() method without any modification.

# Method Overriding



When you write a class that extends another class (or **implements an interface**), and you re-implement methods of the super class (or super interface), it is called overriding.

Method Overriding is achieved when a subclass overrides non-static methods defined in the superclass, following which the **new method** implementation in the subclass that is executed

The new method definition **must have** the same method signature (i.e., method name and parameters) and return type.

Let’s look at a couple of examples.

*Example #1:*

public class Animal { public void move() {

System.out.print("Animal is moving...");

}

}

public class Dog extends Animal {

public void move() {

System.out.print("Dog is running...");

}

}

Here, the Dog class overrides the move() method of the Animal class. In this case, the method in the super class (Animal) is called overridden method. And the method in the sub class (Dog ) is called overriding method.

| **Method Overriding** |
| --- |
| class Base  {  private void show()  {  System.out.println("\n\t This is Show() of Base Class");  }  }  class Child extends Base  {  public void show()//overriding the superclass show()  {  //hide the method of superclass  System.out.println("\n\t This is Overridden show() of Child Class");  }  }  public class MethodOverride\_122  {  public static void main(String[] args)  {  Child c1 = new Child(); c1.show();    }  } |

Program N0.

106

Why Overriding?



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

Use overriding when you want to re-implement (or re-define) a behavior of the super class. For example: the Dog class defines how a dog moves via the move() method. A hound runs faster than a dog so you may need to override the move() method in the Hound class.

Overriding Rules:



**Rule #1: Only inherited methods can be overridden.**

Inheritable methods are declared with the following access modifiers: public, protected and default (in the same package). That means private and default methods (in different package) cannot be overridden.’ **Rule #2: Final and static methods cannot be overridden.**

**Rule #3: The overriding method must have same argument list as the overridden method.**

**Rule #4: The overriding method must have same return type (or sub type).**

In case of the overriding method’s return type is a sub type of the overridden method’s return type, it is called co-variant return type.

**Rule #5: The overriding method must not have more restrictive access modifier.**

For example: if the overridden method is public, you cannot make the overriding method protected, private or default.

**Rule #6: The overriding method must not throw new or broader exceptions.**

In other words, the overriding method may throw fewer or narrower checked exceptions, or any unchecked exceptions.

**Rule #7: Use the super keyword to invoke the overridden method from a sub class.**

**Rule #8: Constructors cannot be overridden.**

**Rule #9: Abstract methods must be overridden by the first concrete (non-abstract) sub class.**

**Rule #10: A static method in a sub class may hide another static one in a super class, and that’s called method hiding.**

**Rule #11: The synchronized modifier has no effect on the rules of overriding.**

The [synchronized](http://www.codejava.net/java-core/the-java-language/synchronized-keyword) modifier relates to the acquiring and releasing of a monitor object in multi-threaded context, therefore it has totally no effect on the rules of overriding. That means a synchronized method can override a nonsynchronized one and vice versa**.**

**Rule #12: The strictfp modifier has no effect on the rules of overriding.**

That means the presence or absence of the [strictfp](http://www.codejava.net/java-core/the-java-language/java-keyword-strictfp) modifier has absolutely no effect on the rules of overriding: it’s possible that a FP-strict method can override a non-FP-strict one and vice-versa.

| **Method overriding Car Example** |
| --- |
| class Car  {  public void move()  {  System.out.println("\n\t This car moves normally");  }  }  class BMW extends Car  {  public void move()  {  System.out.println("\n\t BMW car moves Fast");  }  }  class Ford extends Car  {  public void move()  { |

Program N0. 107

| System.out.println("\n\t Ford Car moves fast and Smooth");  }  }    class xyz extends Car  {  public void move()  {  System.out.println("\n\t xyz Car moves fast and safe");  }  }  public class MethodOverride\_123  {  public static void main(String[] args)  {  BMW b1 = new BMW();  b1.move();    Ford f1 = new Ford();  f1.move();    xyz x1 = new xyz();  x1.move();  }  } |
| --- |

# Difference between method Overloading and Method Overriding



| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is used to increase the readability of the program | Method overriding is used to provide the specific implementation of the method that is already provided by its super class |
| Method overloading is performed within a class | Method overriding occurs in two classes that have IS-A relationship. |
| In case of method overloading parameter must be different | In case of method overriding parameter must be same |

# Abstract Class and Abstract Methods



An ***abstract******class*** is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An ***abstract******method*** is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

**If a class includes abstract methods, then the class itself must be declared abstract, as in:**

public abstract class GraphicObject

{

// declare fields

// declare nonabstract methods

abstract void draw();

}

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

* Abstract modifier means that the class can be used as a superclass only
* Java Abstract classes are used to declare common characteristics of subclasses.
* It can only be used as a superclass for other classes that extend the abstract class.
* Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree
* 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 ▪ Abstract class can have any number of constructors, methods, fields

| **Abstract Class Shape** |
| --- |
| abstract class Shape  {  public double d1, d2;    public Shape(double a, double b)  {  d1 = a; d2 = b;  }    public void show()  {  System.out.println("\n\t d1 = " + d1);  System.out.println("\n\t d2 = " + d2);  }  }  class Rectangle extends Shape  {  public double area;  public Rectangle(double l, double b)  {  super(l, b); |

Program N0. 108

| }    public void calcArea()  {  area = d1\*d2;  show();  System.out.println("\n\t Area of Rectangle :" + area);  }  }  public class AbstractClassDemo\_124  {  public static void main(String[] args)  {  Rectangle rect1 = new Rectangle(12.20,2.50);  rect1.calcArea();  }  } |
| --- |

* Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class

|  |
| --- |
|  |

Program N0.

109

# Abstract Methods



* Methods that are declared **without any body** within an **abstract** **class** is known as abstract method
* The method body will be defined by its subclass
* Abstract method can never be final and static
* Any class that extends an abstract class must implement all the abstract methods declared by the superclass

Syntax abstract class <classname>

{

…fields/instance variables

…Constructors/methods public abstract <ReturnType> methodName (args list); }

| **Abstract method Demo** |
| --- |
| abstract class Shape  {  public double d1, d2;    public Shape(double a)  {  d1 = a; d2 = a;  } |

Program N0. 126

| public Shape(double a, double b)  {  d1 = a; d2 = b;  }  public void show()  {  System.out.println("\n\t d1 = " + d1);  System.out.println("\n\t d2 = " + d2);  }  public abstract void calcArea(); //no definition }  class Rectangle extends Shape  {  public double area;    public Rectangle(double l, double b)  {  super(l, b);  }    public void calcArea() //overriding superclass //abstract method calcArea()  {    area = d1\*d2; show();  System.out.println("\n\t Area of Rectangle = " + area);  }  }  class Circle extends Shape  {  double pi = 3.14; double area;    public Circle(double r)  {  super(r);  }  public void calcArea()  {  area = d1\*d2\*pi;  System.out.println("\n\t Area of Circle = " + area);  }  }  public class AbstractMethodDemo\_126  {  public static void main(String[] args)  {  Rectangle rect1 = new Rectangle(2.20, 4.50); | | |
| --- | --- | --- |
|  |  | rect1.calcArea();  Circle cir1 = new Circle(5.50); cir1.calcArea(); |
| } | } |  |

* Abstract classes are not interfaces
* An abstract class can have 0 or more abstract methods
* When an abstract class with abstract method is extended by a non-abstract class, the abstract methods must be overridden by the subclass

| **Abstract class Demo** |
| --- |
|  |

Program N0.

127

* Abstract methods are usually declared where two ore more subclasses are expected to do a similar thing in different ways through different implementations
* The subclasses (concrete) extend Abstract class and provide different implementations for the abstract methods
* Private methods can not be abstract because they are not visible outside the class
* Static methods can not be abstract. Static methods belong to class, we access static methods using class name instead of object, overriding is only implemented using objects at runtime, hence static method can not be abstract

# Interface



* An interface describes aspects of a class other than those that it inherits from its parent.
* **An interface is a set of requirements that the class must implement.**
* Interface is a pure abstract class
* Interface is syntactically similar to classes, but cannot be instantiated. ▪ The methods in the interface are declared without any body (abstract) ▪ Interface is used to achieve complete abstraction in java.
* Interface must contain abstract methods only

Syntax interface InterfaceName

{

| * **Abstract methods** : Methods that are declared, with no implementation * **Abstract class** : A class with abstract methods, not meant to be instantiated * **Interface** : A named collection of method definitions (without implementations) |
| --- |

constant definitions

method declarations (without implementations) }

* A class implements an interface by doing this:

class SomeClass extends SomeParent implements

interfaceName {

. . .

. . .

}

* A class always extends just one parent but may implement several interfaces.

| **Interface Demo** |
| --- |
| interface DemoInterface  {  public abstract void display();// method without body }  class Demo1 implements DemoInterface  { public void display()  {  System.out.println("\n\t This is Demo1 display()");  }  }  class Demo2 implements DemoInterface  { public void display()  {  System.out.println("\n\t This is Demo2 display()");  }  }  public class InterfaceDemo\_128  { public static void main(String[] args)  { |

Program N0.

128

|  |  | Demo1 d1 = new Demo1(); d1.display();  Demo2 d2 = new Demo2(); d2.display(); |
| --- | --- | --- |
| } | } |  |

* All interface methods must be overridden with public only as overridden method cannot have a weaker access specifier
* Methods inside interface must not be static, final
* All variables declared inside interface are implicitly public static final variables
* All methods declared inside java interfaces are implicitly public and abstract, even if you don’t use public or abstract keyword
* Interface can extend one or more other interface
* As with abstract classes, with interfaces also, objects cannot be created but reference variables can be created.
* Interface reference variable can be assigned with concrete subclass objects. Once assigned, the interface reference variable works like an object.

Difference between an interface and an abstract class?



* An interface cannot implement any methods, whereas an abstract class can
* A class can implement many interfaces but can have only one superclass (abstract or not)
* An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

**Abstract class:**

class Apple extends Food { … }

**Interface:**

public class Person implements Student, Athlete, Chef { … }

| **Abstract class** | **Interface** |
| --- | --- |
| Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes. | Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods. |
| Abstract class is a Class prefix with an abstract keyword followed by Class definition. | Interface is a pure abstract class which starts with interface keyword. |
| Abstract class can also contain concrete methods. | Whereas, Interface contains all abstract methods and final variable declarations. |
| Abstract classes are useful in a situation that  Some general methods should be implemented | Interfaces are useful in a situation that all properties should be implemented. |

and specialization behavior should be implemented by child classes.

# Example



* Let us create a database program for a store. The store sells:
* Goods, each of which has the attributes:
  + description
  + price
* The types of goods are:
  + Food — with an attribute calories. **Food objects are not taxable**
  + Toy — with an attribute minimum age. **Toy objects are taxable**
  + Book — with an attribute author. **Book objects are taxable**
* There are many things that are taxable that are not goods, such as services or entertainment. Also, not all goods are taxable.
* So we want to have the concept **taxable** as a separate concept, not part of the concept of Goods. Here is what the concept Taxable looks like: ▪ A Taxable item,
  + has a taxRate of 5 percent,
  + has a calculateTax() method.
* When implemented in Java, these concepts will appear as classes and an interface.

| | **Concept** | **Parent Class, Child Class, or Interface?** | | --- | --- | | Goods | Parent Class | | Food | Child Class | | Toy | Child Class | | Book | Child Class | | Taxable | Interface | |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

.

| **Interface Example** |
| --- |
| class Goods  {  public String des; public double price;    public Goods(String d, double p)  {  des = d; price = p;  }  public void showGoods()  {  System.out.println("\n\t Description = " + des); |

Program N0. 129

| System.out.println("\n\t Price = " + price);  }  }  interface Taxable  {  public static final double taxRate = 0.05; public abstract double calcTax();  }  class Food extends Goods  {  public int cal;  public Food(String d, double p, int c)  {  super(d, p);  cal = c;  }  public void showFood()  {  showGoods();  System.out.println("\n\t Calories = " + cal);  }  }    class Toy extends Goods implements Taxable  {  public int age;  public Toy(String d, double p, int a)  {  super(d, p); age = a;  }    public double calcTax()  {  return price\*taxRate;  }    public void showToy()  {  showGoods();  System.out.println("\n\t Age = " + age + " years");  System.out.println("\n\t Tax = " + calcTax());  }  }    public class InterfaceDemo\_129  {  public static void main(String[] args)  {  Food jam = new Food("Mix Fruit Jam", 150.00, 200); jam.showFood();    Toy car1 = new Toy("Police Car", 350.00, 5); car1.showToy();    Taxable toy1 = new Toy("Car", 700.00, 5);  Toy t = (Toy)toy1; | | |
| --- | --- | --- |
|  |  | t.showToy(); |
| } | } |  |

# Interface extends interface



interface Singer

{

void sing(); void warmUpVoice();

}

interface Dancer

{

void dance(); void stretchLegs();

} interface Talented extends Singer, Dancer

{

// can sing and dance. Wowwee. }

Where can interfaces are used?



* You can pass an interface as a parameter or assign a class to an interface variable, just like you would to an abstract class.

Example:

Food myLunch = new Sandwich();

Food mySnack = new Apple();

Student amit = new Person();

//assuming that Person implements Student

* If Person has methods eat(Food f) and teach(Student s), the following is possible:

Person sumit= new Person(); sumit.teach(amit); sumit.eat(myLunch);

# Wrapper Class



In Java programming language, there are eight primitive types and each of these has a corresponding library class of reference type. For example, there is a class **java.lang.Integer** that corresponds to primitive type int.

These kinds of classes are called wrappers. The wrapper classes are immutable; we cannot change a wrapped value after the wrapper has been constructed. They are also final, so we cannot subclass them. Why do we have wrapper classes for primitive types?

* The wrapper classes are used whenever a primitive type needs to be treated as an Object. For example, the classes in the Collections API (like ArrayList) can only hold Object references.
* To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

# Boxing and Unboxing in Java



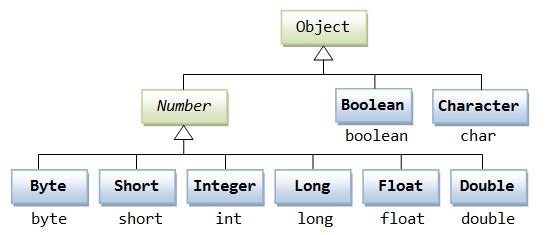
Conversion of a primitive type to the corresponding reference type is called **boxing**, such as an int to a java.lang.Integer.

**Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an **int** to an **Integer**, a **double** to a **Double**, and so on.

Conversion of the reference type to the corresponding primitive type is called **unboxing**, such as Byte to byte. Since JDK 1.5, Conversion from primitive types to corresponding wrapper objects and vice versa can happen automatically.

The Java compiler applies **autoboxing** when a primitive value is:

* Passed as a parameter to a method that expects an object of the corresponding wrapper class.
* Assigned to a variable of the corresponding wrapper class.



Integer intObj = 5566; // autobox from int to Integer int i = intObj; // auto-unbox from Integer to int Double doubleObj = 55.66; // autoboxing from double to Double double d = doubleObj; // atuo-unbox from Double to double

# Collection Framework



Although we can use an array to store a group of elements of the same type (either primitives or objects). The array, however, does not support so-called *dynamic allocation* - it has a ***fixed length*** which cannot be changed once allocated. Furthermore, array is a simple linear structure. Many applications may require more complex data structure such as linked list, stack, hash table, sets, or trees.

In Java, dynamically allocated data structures (such as ArrayList, LinkedList, Vector, Stack, HashSet, HashMap, Hashtable) are supported in a unified architecture called the Collection Framework, which mandates the common behaviors of all the classes.

A collection, as its name implied, is simply an object that holds a collection (or a group, a container) of objects. Each item in a collection is called an element. A framework, by definition, is a set of interfaces that force you to adopt some design practices. A well-designed framework can improve your productivity and provide ease of maintenance.

In terms of programming, a collection is a data structure that holds a set of objects in a specific manner. It looks like arrays but collections are more advanced and more flexible. An array simply stores a fixed number of objects, whereas a collection stores objects dynamically, i.e. you can add or remove objects as you wish.

**Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

**Java Collections Framework** is a set of reusable data structures and algorithms which are designed to free programmers from implementing data structures themselves so that they can focus on business logics.

The Java Collections Framework provides common data structures implementations which are enough for generalpurpose such as list, set, map, queue, tree, etc. These collections are high-performance, high-quality, and easy to use with very good documentation.

In addition, the Java Collections Framework provides useful and robust algorithms such as searching and sorting on collections, and the interoperability between collections and arrays.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

## Collection Framework

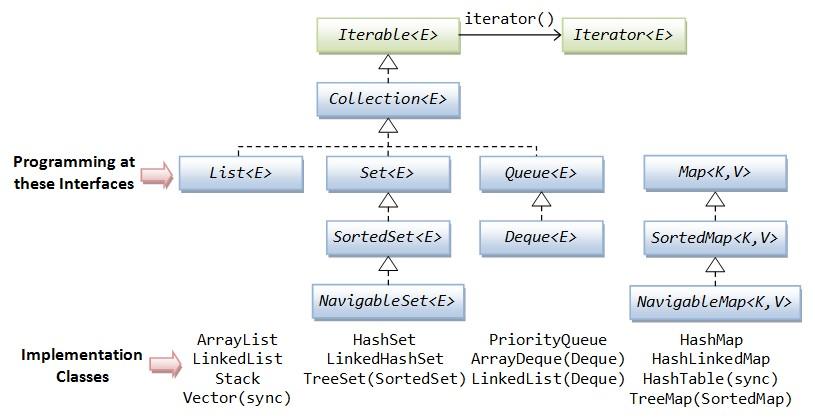
The collection framework provides a unified interface to store, retrieve and manipulate the elements of a collection, regardless of the underlying and actual implementation. This allows the programmers to program at the interfaces, instead of the actual implementation.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc)

# Interfaces



The **core collection interfaces** encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



## Iterable<E> Interface

The Iterable<E> interface, which takes a generic type E and read as Iterable of element of type E, declares one abstract method called iterator() to retrieve the Iterator<E> object associated with all the collections. ThisIterator object can then be used to transverse through all the elements of the associated collection.

**Iterator<E> iterator();** // Returns the associated Iterator instance // that can be used to transverse thru all the elements of the collection

All implementations of the collection (e.g., ArrayList, LinkedList, Vector) must implement this method, which returns an object that implements Iterator interface.

## Iterator<E> Interface

The Iterator<E> interface, declares the following three abstract methods: **boolean hasNext()** // Returns true if it has more elements **E next()** // Returns the next element of generic type E **void remove()** // Removes the last element returned by the iterator

As seen in the introductory example, you can use a while-loop to iterate through the elements with the Iterator as follows:

**List<String> lst = new ArrayList<String>(); lst.add("alpha"); lst.add("beta"); lst.add("charlie");**

// Retrieve the Iterator associated with this List via the iterator() method **Iterator<String> iter = lst.iterator();**

// Transverse thru this List via the Iterator **while (iter.hasNext())**  // Retrieve each element and process

**{**

**String str = iter.next();**

**System.out.println(str);**

}

## Collection Interface

The Collection<E>, which takes a generic type E and read as Collection of element of type E, is the root interface of the Collection Framework. It defines the common behaviors expected of all classes, such as how to add or remove an element, via the following abstract methods:

The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

## List<E>

List Interface Models a resizable linear array, which allows indexed access. List can contain duplicate elements.

Frequently-used implementations of List include ArrayList, LinkedList, Vector and Stack

## Set<E>

Itmodels a mathematical set, where no duplicate elements areallowed. Frequently-used implementations of Set are HashSet and LinkedHashSet. The sub-interface SortedSet<E> models an ordered and sorted set of elements, implemented by TreeSet.

## Queue<E>

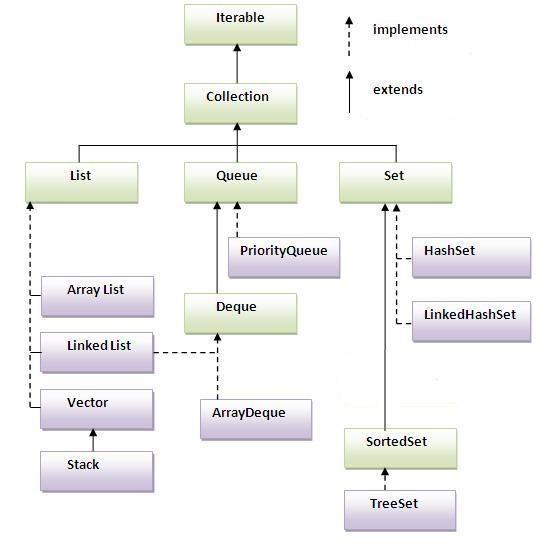
Queue<E> interface models queues such as First-in-First-out (FIFO) queue and priority queue. It sub

interface Deque<E> models queues that can be operated on both ends. Implementations include PriorityQueue, ArrayDeque and LinkedList.

## Map**<K,V>**,

The interface Map<K,V>, which takes two generic types K and V and read as Map of Key type K and Value type V, is used as a collection of "key-value pairs". No duplicate key is allowed. Frequently-used implementations include

HashMap, Hashtable and LinkedHashMap. Its sub-interface SortedMap<K, V> models an ordered and sorted map, based on its key, implemented in TreeMap.



# Methods of Collection interface



There are many methods declared in the Collection interface. They are as follows:

| **No. Method Description** | | | | | |
| --- | --- | --- | --- | --- | --- |
| 1 | | public boolean **add**(Object element) | | is used to insert an element in this collection. | |
| 2 | | public boolean **addAll**(Collection c) | | is used to insert the specified collection elements in the invoking collection. | |
| 3 | | public boolean **remove**(Object element) | | is used to delete an element from this collection. | |
| 4 | | public boolean **removeAll**(Collection c) | | is used to delete all the elements of specified collection from the invoking collection. | |
| 5 | | public boolean **retainAll**(Collection c) | | is used to delete all the elements of invoking collection except the specified collection. | |
| 6 | | public int **size**() | | return the total number of elements in the collection. | |
| 7 | | public void **clear**() | | removes the total no of element from the collection. | |
| 8 | | public boolean **contains**(Object element) | | is used to search an element. | |
| 9 | | public boolean **containsAll**(Collection c) | | is used to search the specified collection in this collection. | |
| 10 | | public Iterator **iterator**() | | returns an iterator. | |
| 11 | | public Object[] toArray() | | converts collection into array. | |
|  | 12 | | public boolean isEmpty() | | checks if collection is empty. | |
|  | 13 | | public boolean equals(Object element) | | matches two collection. | |
|  | 14 | | public int hashCode() | | returns the hashcode number for collection. | |

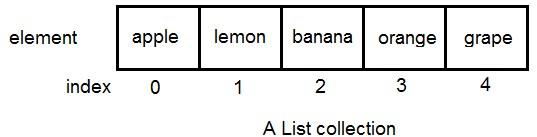
What is a List?



A **List** is a kind of collections in the Java Collection Framework. It’s used widely in Java programming and programmers love it.

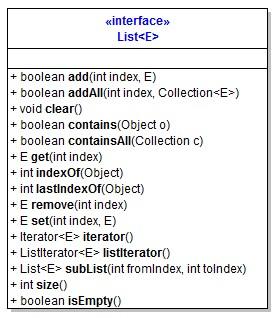
A **List** collection maintains elements in form of index-based, meaning the first element is stored at 0-index, the second one is at 1-index, the third one is at 2-index, and so on.

A list collection stores elements by insertion order (either at the end or at a specific position in the list). A list maintains indices of its elements so it allows adding, retrieving, modifying, removing elements by an integer index (zero-based index; the first element is at 0-index, the second at 1-index, the third at 2-index, and so on). The following picture illustrates a list that stores some String elements:



A list can store objects of any types. Primitive types are automatically converted to corresponding wrapper types, e.g. integer numbers are converted to Integer objects. It allows null and duplicate elements, and orders them by their insertion order (index).

The following class diagram depicts the primary methods defined in the **java.util.List** interface:

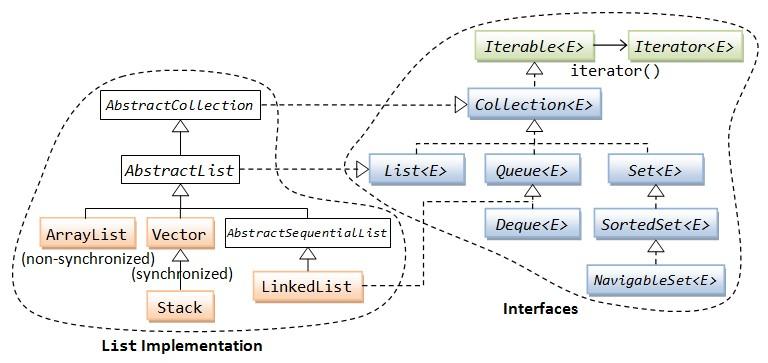


Characteristics of Lists:



A **List** collection can store any objects. It maintains elements by insertion order, meaning that when you add an object to a list, the object will be placed at the end of the list.

A **List** allows null and duplicate elements, and orders them by their insertion order, hence most operations on list are based on the indices.



List Implementations:



The Java Collection Framework provides two major implementations of the List interface. They are **ArrayList** and **LinkedList**. That means List is the super interface, and **ArrayList** and **LinkedList** are two sub classes.

## ArrayList

An implementation that stores elements in a backing array. The array’s size will be automatically expanded if there isn’t enough room when adding new elements into the list. It’s possible to set the default size by specifying an initial capacity when creating a new **ArrayList**.

* Java ArrayList class uses a **dynamic array (Resizable Array Data Structure)** for storing the elements. It extends **AbstractList** class and implements List interface.
* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

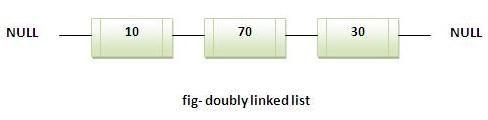
Basically, an **ArrayList** offers constant time for the following

operations: **size**, **isEmpty**, **get**, **set**, **iterator**, and **listIterator**; amortized constant time for the **add** operation; and linear time for other operations. Therefore, this implementation can be considered if we want fast, random access of the elements.

## LinkedList

An implementation that stores elements in a doubly-linked list data structure. It offers constant time for adding and removing elements at the end of the list; and linear time for operations at other positions in the list. Therefore, we can consider using a **LinkedList** if fast adding and removing elements at the end of the list is required.

* Java LinkedList class uses **doubly linked list** to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
* Java LinkedList class can be used as list, stack or queue.



Besides **ArrayList** and **LinkedList**, Vector class is a legacy collection and later was retrofitted to implement the **List** interface. **Vector** is thread-safe, but **ArrayList** and **LinkedList** are not. The following class diagram depicts the inheritance tree of the **List** collections:

## Vector

Vector is also a List implementation. However, **Vector** is an old collection which was created before the Java Collection Framework. Nowadays, **Vector** is obsolete, and it exists only for the purpose of backward compatibility with old APIs.

Why and When Use Lists?



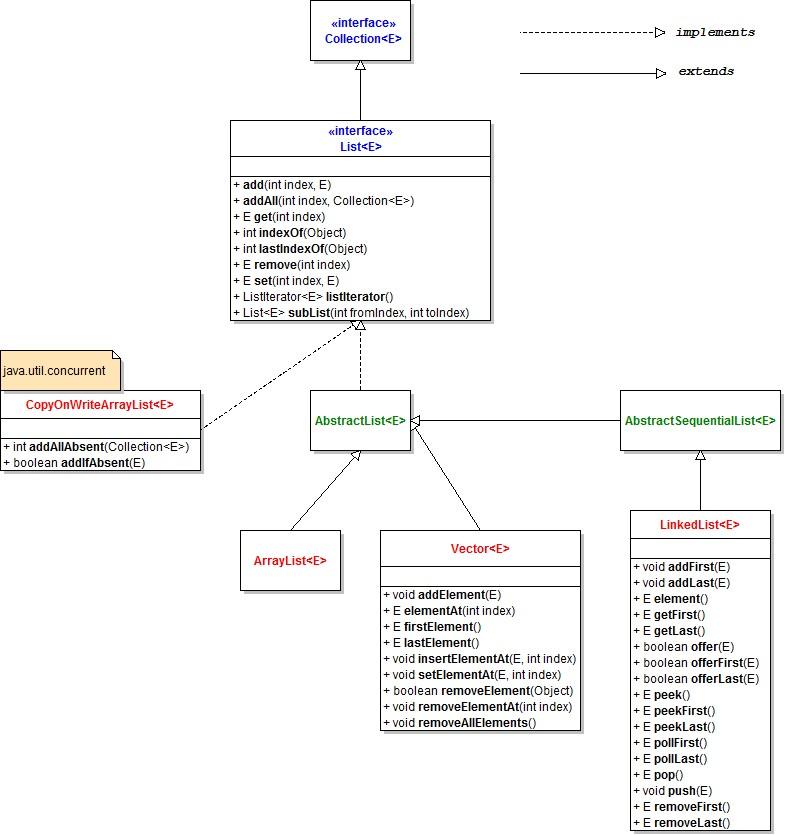
Consider using a List collection when you want to store and retrieve elements based on index.

More specifically, consider using an **ArrayList** when you want fast, random accessing of elements as an **ArrayList** provides constant time access to any elements in the list.

And consider using a **LinkedList** when you want fast adding and removing elements at the end of the list, as a **LinkedList** store elements in a doubly-linked data structure.

* **List**<E> is the base interface for all kinds of list. It defines general operations for a List type.
* Abstract subclasses: **AbstractList<E>** and **AbstractSequentialList**<E>
* Concrete implementation classes: **ArrayList<E>, Vector<E>**, **LinkedList<E>** and **CopyOnWriteArrayList<E>(**this class is under **java.util.concurrent** package).
* Legacy collection: **Vector<E>**
* Implementation classes in JDK which are not members of Java Collections Framework: **AttributeList**,**RoleList**, **RoleUnresolvedList** and **Stack**.

**The following class diagram describes the hierarchy structure of List API in Java Collections Framework:**



# 1. Creating a new list



It’s a good practice to declare a list instance with a generic type parameter, for example:

List<Object> listAnything = new ArrayList<Object>();

List<String> listWords = new ArrayList<String>();

List<Integer> listNumbers = new ArrayList<Integer>();

List<String> linkedWords = new LinkedList<String>();

Since Java 7, we can remove the type parameter on the right side as follows:

List<Integer> listNumbers = new ArrayList<>();

List<String> linkedWords = new LinkedList<>();

The compiler is able to infer the actual type parameter from the declaration on the left side.

When creating a new ArrayList using the empty constructor, the list is constructed with an initial capacity of ten. If you are sure how many elements will be added to the list, it’s recommended to specify a capacity which is large enough.

Let’s say, if we know that a list contains around 1000 elements, declare the list as follows:

List<Integer> listNumbers = new ArrayList<>(1000);

It’s also possible to construct a list that takes elements from an existing collection, for example:

List<Integer> listNumberOne; // existing collection

List<Integer> listNumberTwo = new ArrayList<>(listNumberOne);

The listNumberTwo constructed with copies of all elements from the listNumberOne.

# 2. Basic operations: adding, retrieving, updating, removing elements



## Adding elements

The methods **add(Object)**, **add(index, Object)** and **addAll(Collection)** are used to add elements to the list. It requires to add elements of the same type (or sub type) as the type parameter declared by the list. For example:

List<String> listStrings = new ArrayList<String>(); listStrings.add("One"); listStrings.add("Two"); listStrings.add("Three");

// But this will cause compile error

listStrings.add(123);

Adding elements of sub types of the declared type:

List<Number> linkedNumbers = new LinkedList<>(); linkedNumbers.add(new Integer(123)); linkedNumbers.add(new Float(3.1415)); linkedNumbers.add(new Double(299.988)); linkedNumbers.add(new Long(67000));

We can insert an element into the list at a specified index, for example:

listStrings.add(1, "Four");

That inserts the String “Four” at the 2nd position in the list.

We can also add all elements of an existing collection to the end of the list:

listStrings.addAll(listWords);

Or add the elements to the list at a specified position:

listStrings.addAll(2, listWords);

That inserts all elements of the listWords collection at 3rd position of the listStrings collection.

## Retrieving elements

The **get(index)** method is used to retrieve an element from the list at a specified index. For example, the following code gets an element at 2ndposition in the array list and an element at 4th position in the linked list:

String element = listStrings.get(1);

Number number = linkedNumbers.get(3);

For a **LinkedList** implementation, we can get the first and the last elements like this:

LinkedList<Number> numbers = new LinkedList<Number>();

// add elements to the list...

// get the first and the last elements:

Number first = numbers.getFirst();

Number last = numbers.getLast();

Note that the **getFirst**() and **getLast**() methods are specific to the LinkedList class.

## Updating elements

Use the **set(index, element)** method to replace the element at the specified index by the specified element. For example:

listStrings.set(2, "Hi");

That replaces the 3rd element in the list by the new String “Hi”.

## Removing elements

To remove an element from the list, use the **remove(index)** or **remove(Object)** method which removes the element at the specified index or by object reference. For example:

Remove the element at the 3rd position in the list:

listStrings.remove(2);

If the specified index is out of range (index < 0 or index >= list size), a java.lang.IndexOutOfBoundsException is thrown.

Remove the String element “Two” in the list:

listStrings.remove("Two");

Notes about the remove(Object) method:

* It compares the specified object with the elements in the list using their equals() method, so if you use your own defined object type, make sure it implements the equals() method correctly.
* It only removes the first occurrence of the specified element in the list (i.e. if a list contains duplicate elements, only the first element is removed).
* It returns true if the list contained the specified element, or false otherwise. Thus it’s recommended to check return value of this method, for example:

if (listStrings.remove("Ten")) { System.out.println("Removed");

} else {

System.out.println("There is no such element"); }

To remove all elements in the list, use the **clear()** method:

listStrings.clear();

## 3. Iterating over a list

Basically, we can use the enhanced for loop to iterate through all elements in the list, as follows:

for (String element : listStrings) {

System.out.println(element);

}

Or use an iterator like this:

Iterator<String> iterator = listStrings.iterator(); while (iterator.hasNext()) {

System.out.println(iterator.next()); }

For more list-specific, use a list iterator as shown below:

Iterator<Number> iterator = linkedNumbers.listIterator();

while (iterator.hasNext()) {

System.out.println(iterator.next()); }

Since Java 8, we can use the forEach() method like this:

listStrings.forEach(s -> System.out.println(s));

# 4. Searching for an element in a list



To search for position of a specific element in the list or to know if the list contains the specified element, the following methods can be used:

* **boolean contains(Object)**: returns true if the list contains the specified element.
* **int indexOf(Object)**: returns the index of the first occurrence of the specified element in the list, or -1 if the element is not found.
* **int lastIndexOf(Object)**: returns the index of the last occurrence of the specified element in the list, or -1 if the element is not found.

Examples:

if (listStrings.contains("Hello")) {

System.out.println("Found the element");

} else {

System.out.println("There is no such element");

}

int firstIndex = linkedNumbers.indexOf(1234); int lastIndex = listStrings.indexOf("Hello");

Note that the above methods compare the elements using their **equals()** method, so if you define your own type, make sure it implements the equals() method correctly.

# 5. Sorting a list



The simplest way to sort out elements in a list is using the **Collections.sort()** static method which sorts the specified list into ascending order, based on the natural ordering of its elements. Here’s an example:

List<String> listStrings = new ArrayList<String>(); listStrings.add("D"); listStrings.add("C"); listStrings.add("E"); listStrings.add("A"); listStrings.add("B");

System.out.println("listStrings before sorting: " + listStrings);

Collections.sort(listStrings);

System.out.println("listStrings after sorting: " + listStrings); Output:

listStrings before sorting: [D, C, E, A, B] listStrings after sorting: [A, B, C, D, E]

Note that all elements in the list must implement the Comparable interface, so if you define your own type, make sure it implements that interface and its **compareTo()** method.

# 6. Copying one list into another



The **Collections.copy(dest, src)** static method allows us to copy all elements from the source list into the destination one. Note that the destination list must be large enough to contain the entire source list. Here’s an example:

List<String> sourceList = new ArrayList<String>(); sourceList.add("A"); sourceList.add("B"); sourceList.add("C"); sourceList.add("D");

List<String> destList = new ArrayList<String>(); destList.add("V"); destList.add("W"); destList.add("X"); destList.add("Y"); destList.add("Z");

System.out.println("destList before copy: " + destList);

Collections.copy(destList, sourceList);

System.out.println("destList after copy: " + destList); The output would be:

destList before copy: [V, W, X, Y, Z] destList after copy: [A, B, C, D, Z]

# 7. Shuffling elements in a list



To randomly permute elements in a list, use the **Collections.shuffle()** static method. Here’s a quick example:

List<Integer> numbers = new ArrayList<Integer>(); for (int i = 0; i <= 10; i++) numbers.add(i);

System.out.println("List before shuffling: " + numbers);

Collections.shuffle(numbers);

System.out.println("List after shuffling: " + numbers);

The output would be:

List before shuffling: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

List after shuffling: [6, 4, 5, 0, 1, 3, 9, 7, 2, 10, 8]

# 8. Reversing elements in a list



To reverse order of elements in a list, use the **Collections.reverse()** static method. Here’s a quick example:

List<Integer> numbers = new ArrayList<Integer>(); for (int i = 0; i <= 10; i++) numbers.add(i);

System.out.println("List before reversing: " + numbers);

Collections.reverse(numbers);

System.out.println("List after reversing: " + numbers);

The output would be:

List before reversing: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

List after reversing: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# 9. Extracting a portion of a list



The **subList(fromIndex**, **toIndex)** allows us to get a portion of the list between the specified **fromIndex** (inclusive) and **toIndex** (exclusive). Here’s an example:

List<String> listNames = Arrays.asList("Tom", "John", "Mary", "Peter", "David", "Alice");

System.out.println("Original list: " + listNames);

List<String> subList = listNames.subList(2, 5);

System.out.println("Sub list: " + subList);

Output:

Original list: [Tom, John, Mary, Peter, David, Alice]

Sub list: [Mary, Peter, David]

Note that the sub list is just a view of the original list, so any modifications made on the original list will reflect in the sub list.

# 10. Converting between Lists and arrays



The Java Collection Framework allows us to easily convert between lists and arrays.

The **Arrays.asList(T… a)** method converts an array of type **T** to a list of type **T**. Here’s an example:

List<String> listNames = Arrays.asList("John", "Peter", "Tom", "Mary", "David", "Sam");

List<Integer> listNumbers = Arrays.asList(1, 3, 5, 7, 9, 2, 4, 6, 8);

System.out.println(listNames);

System.out.println(listNumbers);

Output:

[John, Peter, Tom, Mary, David, Sam]

[1, 3, 5, 7, 9, 2, 4, 6, 8]

And the **List** interface provides the **toArray()** method that returns an array of Objects containing all of the elements in the list in proper sequence (from first to last element). Here’s an example:

List<String> listWords = new ArrayList<String>();

// add elements to the list

Object[] arrayWords = listWords.toArray();

And the **toArray(T[] a)** method returns an array of type **T**, for example:

String[] words = listWords.toArray(new String[0]);

Integer[] numbers = listNumbers.toArray(new Integer[0]);

Note that the returned array contains copies of elements in the list, that means we can safely modify the array without affecting the list.

# 11. Concurrent lists



By default, **ArrayList** and **LinkedList** are not thread-safe, so if you want to use them in concurrent context, you have to synchronize them externally using the **Collections.synchronizedList()** static method which returns a synchronized list that wraps the specified list. For example:

List<Object> unsafeList = new ArrayList<Object>();

List<Object> safeList = Collections.synchronizedList(unsafeList);

Note that you must manually synchronize the returned list when iterating over it, for example:

synchronized (safeList) {

Iterator<Object> it = safeList.iterator(); while (it.hasNext()) {

System.out.println(it.next());

}

}

# Difference between ArrayList and LinkedList



ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

| **ArrayList** | **LinkedList** |
| --- | --- |
| ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and **Deque interfaces**. |
| ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# Java Non-generic Vs Generic Collection



Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

1. ArrayList al=**new** ArrayList(); //creating old non-generic arraylist Let's see the new generic example of creating java collection.
2. ArrayList<String> al=**new** ArrayList<String>(); //creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

# Vector



* Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
* **Vector is synchronized**
* **Vector contains many legacy methods that are not part of the collections framework** ▪ Constructors

Vector( )

Vector(int size)

Vector(int size, int incr)

Vector(Collection c)

* Vector Class allows duplicate values
* It is similar to ArrayList, internally follows Resizable Array Structure

| Important Methods |  |
| --- | --- |
| boolean add (Object o) | Add element to the Vector |
| void add (int index, Object element) | Add element to the Vector at specified position |
| void clear () | Clears elements |
| Object get (int index) | Returns element at specified index |
| boolean remove (Object o) | Removes element |
| int size() | Returns size |
| Void addElement() | Add element to the Vector |
| Object elementAt(int index) | Return the element at specified index |
| Enumeration elements() | Return an enumeration of element in vector |
| Object firstElement() | Return first element in the Vector |
| Object lastElement() | Return last element in the Vector |
| boolean removeAllElement() | Remove all element of the Vector |
| **Vector Demo** |
| import java.util.\*;    public class VectDemo\_104  {  public static void main(String[] args)  {  Vector v = new Vector(); v.add(10);  v.add(23.43);  v.add("Hello World");  v.add(10);    System.out.println("\n\t Elements = " + v); System.out.println("\n\t Using For :"); for(int i=0; i<v.size(); i++)  {  System.out.print("\t" + v.get(i));  }    System.out.println("\n\t Using Iterator :");  Iterator it = v.iterator();  while(it.hasNext())  {  System.out.print("\t" + it.next());  }      System.out.println("\n\t Using ListIterator :");  ListIterator lit = v.listIterator();  while(lit.hasNext())  {  System.out.print("\t" + lit.next());  } |

Program N0. 104

|  |  | System.out.println("\n\tDisplay elements reverse :"); while(lit.hasPrevious())  {  System.out.print("\t" + lit.previous()); } |
| --- | --- | --- |
| } | } |  |



# Stack



* Stack is a subclass of Vector that implements a standard **last-in, first-out** stack.
* Stack only defines the default constructor, which creates an empty stack.

Stack( )

* Stack includes all the methods defined by Vector, and adds several of its own
* Stack Class is similar to Vector [It is Synchronized]
* It follows Stack Data Structure

| **SN** | **Methods with Description** |
| --- | --- |
| 1 | **boolean empty()**  Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements. |
| 2 | **Object peek( )**  Returns the element on the top of the stack, but does not remove it. |
| 3 | **Object pop( )**  Returns the element on the top of the stack, removing it in the process. |
| 4 | **Object push(Object element)**  Pushes element onto the stack. element is also returned. |
| 5 | **int search(Object element)**  Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned. |

| **Stack Demo** |
| --- |
| import java.util.\*;    public class StackDemo\_105  {  public static void main(String[] args)  {  Stack st = new Stack(); st.push(10); st.push(34.32); st.push("Hello World");    System.out.println("\n\t Elements = " + st);    System.out.print("\n\t Element at Top = " + st.pop());    System.out.println("\n\t Elements = " + st);  }  } |

Program N0.

105

# Summary of List Implementations



| **Property** | **ArrayList** | **LinkedList** | **Vector** | **Stack** |
| --- | --- | --- | --- | --- |
| Ordered | Ordered by index | Ordered by index | Ordered by index | Ordered by Index |
| Null Values | Allowed | Allowed | Allowed | Allowed |
| Duplicate | Allowed | Allowed | Allowed | Allowed |
| Synchronized | No | No | Yes | Yes |
| Initial Capacity | 10 | Not Applicable | 10 | 10 |
| Data Structure | Resizable Array | Doubly Linked List | Resizable Array | Resizable Array |

# 4 Mechanisms of Collections Iteration



Iteration is one of the basic operations carried on a collection. Basically, an iteration takes elements from a collection one after another, from the first element to the last one.

For example, you may want to go through all students in a class to print their names or to find who have high scores in the recent exam. Or you may want to traverse through a list of numbers to calculate the sum and average. Such kinds of operations are very common in programming.

The Java programming language provides four mechanisms for iterating over collections, including for loops, iterator and forEach (since Java 8).

Before going to each kind of iteration, suppose that we have a List collection as follows:

List<String> listNames = new ArrayList<>(); listNames.add("Tom"); listNames.add("Mary"); listNames.add("Peter"); listNames.add("John"); listNames.add("Kim");

This list contains names of all students in a class. Note that the diamond operator <> used in the right side of the assignment:

ArrayList<>();

This syntax can be used from Java 7, which allows us to declare generics collections in a more compact way, as the compiler can infer the parameter type in the right side from the left side (thus the so-called ***type inference***).

1. The Classic For Loop:



This iteration mechanism is very familiar in programming in which a counter variable runs from the first element to the last one in the collection. Here’s the code that iterates over the listNames above:

for (int i = 0; i < listNames.size(); i++) {

String aName = listNames.get(i);

System.out.println(aName);

}

**Here are the pros of this mechanism:**

* + This is the most familiar construct in programming.
  + Useful if we need to access and use the counter variable, e.g. print the numeric order of the students: 1, 2, 3, etc.

**And here are the cons:**

* + Using a counter variable requires the collection must store elements in form of index-based like ArrayList, and we must know the collection’s size beforehand.
  + The collection must provide a method to access its elements by index-based, which not supported by all collections, e.g. a Set does not store elements as index-based. Thus this mechanism cannot be used with all collections.

1. The Iterator Mechanism:



Due to the limitations of the classic for loop, the **Iterator** mechanism is created to allow us to iterate all kinds of collections. Thus you can see the **Collection** interface defines that every collection must implement the **iterator**() method.

## Concept of the Iterator

An iterator is an object that enables us to traverse a collection. There is an iterator (**java.util.Iterator**) in all the top level interfaces of the Java Collections Framework that inherits java.util.Collection interface. These interfaces are **java.util.List**, **java.util.Queue**, **java.util.Deque**, and **java.util.Set**. Furthermore, there is the **java.util.Map** interface that does not inherit **java.util.Collection**.

Lists also have a special iterator called a list iterator (**java.util.ListIterator**). What’s the difference?

The **java.util.Iterator** is forward looking only while the **java.util.ListIterator** is bidirectional (forward and backward). Furthermore, the **java.util.ListIterator** inherits **java.util.Iterator**. The result of using either iterator to loop through a list will be the same as we will see later.

The following example explains the concept of iterator:

Iterator<String> iterator = listNames.iterator(); while (iterator.hasNext()) {

String aName = iterator.next();

System.out.println(aName);

}

This code snippet does the same thing as the classic for loop example above. You may need some explanations:

* The hasNext() method returns true if the collection has more elements to traverse, otherwise return false.
* The next() method returns the current element. Note that we don’t have to cast the returned object as we use generics.

Here’s another example that illustrates how to iterate over a Set using an Iterator:

Set<Integer> numbers = new HashSet<>();

numbers.add(100); numbers.add(35); numbers.add(89); numbers.add(71);

Iterator<Integer> iterator = numbers.iterator(); while (iterator.hasNext()) {

Integer aNumber = iterator.next();

System.out.println(aNumber); }

And here’s another example demonstrating how to iterate over a Map using an iterator:

Map<Integer, String> mapAscii = new HashMap<>(); mapAscii.put(65, "A"); mapAscii.put(66, "B"); mapAscii.put(67, "C"); mapAscii.put(68, "D");

Iterator<Integer> keyIterator = mapAscii.keySet().iterator(); while (keyIterator.hasNext()) {

Integer key = keyIterator.next();

String value = mapAscii.get(key);

System.out.println(key + " -> " + value); }

Because the map stores elements in form of key=value pairs, first we need to get the iterator of the keys (a Set collection), then use this iterator to get each key, and retrieve the value corresponds to that key.

3. The Enhanced For Loop:



Since Java 5, programmers can use a more succinct syntax to iterate over a collection - It’s the ***enhanced for loop***.

For example, the following code uses the enhanced for loop to iterate over the listNames collection above:

for (String aName : listNames) {

System.out.println(aName);

}

The code is more compact and more readable. That’s why this construct is called enhanced for loop - an enhanced feature of the Java programming language.

NOTE:

The enhanced for loop actually uses an iterator behind the scenes. That means the Java compiler will convert the enhanced for loop syntax to iterator construct when compiling. The new syntax just gives the programmers a more convenient way for iterating over collections.

Using the enhanced for loop, we can re-write the code to iterate the Set collection above like this:

for (Integer aNumber : numbers) {

System.out.println(aNumber); }

Compare to the previous code (using iterator), this code is incredible simpler and more understandable right?

And the code that iterates over a Map can be re-written using the enhanced for loop like this:

for (Integer key : mapAscii.keySet()) {

String value = mapAscii.get(key);

System.out.println(key + " -> " + value); }

This looks much simpler than the previous code using iterator, right? Thanks to the enhanced for loop - it helps programmers write code more quickly and more readable.

As the Java programming language evolves, we have a new mechanism which is describe below.

4. The forEach Mechanism:



Java 8 with Lambda expressions, introduces a totally new way for iterating over collections - it’s the ***forEach*** mechanism.

What’s the biggest difference between the **forEach** mechanism and the previous ones?

Well, in the previous mechanisms (classic for loop, iterator and enhanced for loop), the programmers control how the collection is iterated. The iteration code is not part of the collection itself - it’s written by programmers - hence the term ***external iteration***.

In contrast, the new mechanism encapsulates the iteration code in the collection itself, thus the programmers do not have to write code for iterating collections. Instead, the programmers specify what-to-do in each iteration - this is the big difference! Hence the term ***internal iteration***: the collections handle the iteration itself, whereas the programmers pass the action - what needs to do in each iteration.

The following example helps you understand the concepts:

listNames.forEach(name -> System.out.println(name));

Amazing! This code looks even more compact and more readable than the enhanced for loop version. As we can read the above line like this: for each element in the list Names, print the name to the console.

Since Java 8, each collection has a forEach() method that implements the iteration internally. Note that this method takes a Lambda expression or in other words, the programmers can pass their code - or function - into this method. As shown in the above example, the code to print each element is passed into the method.

What is Set?



Basically, **Set** is a type of collection that does not allow duplicate elements. That means an element can only exist once in a **Set**. It models the set abstraction in mathematics.

Characteristics of a Set collection:



The following characteristics differentiate a Set collection from others in the Java Collections framework:

* Duplicate elements are not allowed.
* Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a **Set**.

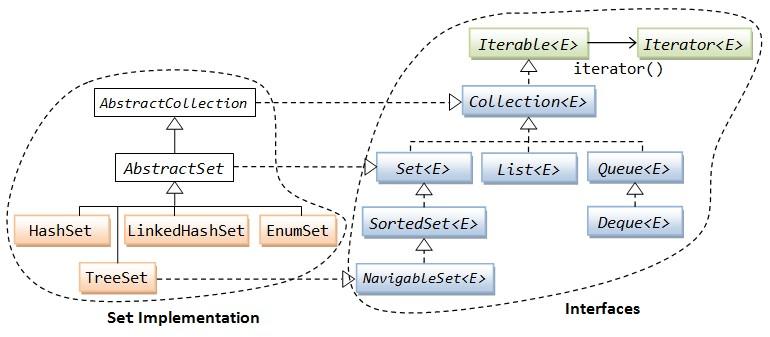
Why and When Use Sets?

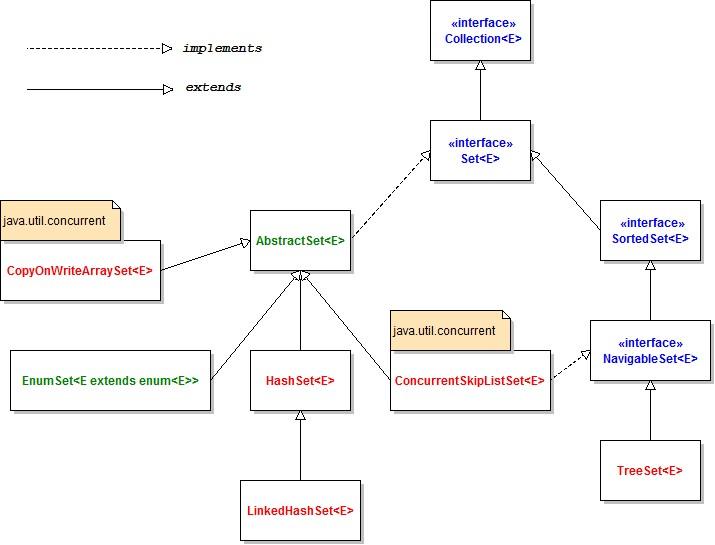


Based on the characteristics, consider using a **Set** collection when:

* You want to store elements distinctly without duplication, or unique elements. ▪ You don’t care about the order of elements.

For example, you can use a **Set** to store unique integer numbers; you can use a **Set** to store cards randomly in a card game; you can use a Set to store numbers in random order, etc.





Set Implementations:



The Java Collections Framework provides three major implementations of the Set interface: **HashSet**, **LinkedHashSet** and **TreeSet**.

## HashSet

HashSet is the best-performing implementation and is a widely-used **Set** implementation. It represents the core characteristics of sets: no duplication and unordered.

* Uses hashtable to store the elements.
* Contains unique elements only.
* HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.
* Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a **Set**
* A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
* The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically ▪ The first form constructs a default hash set:

HashSet( )

* The following constructor form initializes the hash set by using the elements of c

HashSet(Collection c)

* The following constructor form initializes the capacity of the hash set to capacity. The capacity grows automatically as elements are added to the Hash.

HashSet(int capacity)

| **SN** | **Methods with Description** |
| --- | --- |
| 1 | **boolean add(Object o)**  Adds the specified element to this set if it is not already present. |
| 2 | **void clear()**  Removes all of the elements from this set. |
| 4 | **boolean contains(Object o)**  Returns true if this set contains the specified element |
| 5 | **boolean isEmpty()**  Returns true if this set contains no elements. |
| 7 | **boolean remove(Object o)**  Removes the specified element from this set if it is present. |
| 8 | **int size()**  Returns the number of elements in this set (its cardinality). |

| **HashSet Demo** |
| --- |
| import java.util.\*;    public class HashSetDemo\_106  {  public static void main(String[] args)  {  HashSet set1 = new HashSet();  set1.add(99); set1.add(10); set1.add(543.34); set1.add("Hello World"); set1.add(new Integer(23)); //boolean result = set1.add(99);  System.out.println(set1.add("Hello World"));  System.out.println("\n\t Elements = " + set1);  }  } |

Program N0.

106

## LinkedHashSet

This implementation orders its elements based on insertion order. So consider using a **LinkedHashSet** when you want to store unique elements in order.

* This class extends **HashSet**, but adds no members of its own.
* LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted
* The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.
* The LinkedHasSet internally follow hashing technique and DoubleLinkedList Structure
* The LinkedHashSet class supports following constructors

LinkedHashSet( )

LinkedHashSet(Collection c)

* Creation of LinkedHashSet

LinkedHashSet lset = new LinkedHashSet()

| **LinkedHashSet Demo** |
| --- |
| import java.util.\*;    public class LinkedHashSetDemo\_107  {  public static void main(String[] args)  {  LinkedHashSet hs = new LinkedHashSet();  hs.add(10); hs.add(43); hs.add("Hello World");  hs.add(10);    System.out.println("\n\t Elements = " + hs); }  } |

LinkedHashSet<E> lset = new LinkedHashSet<E>(); Program N0.

107

## TreeSet

This implementation orders its elements based on their values, either by their natural ordering, or by a **Comparator** provided at creation time.

* TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
* Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

| **Treeset Demo** |
| --- |
| import java.util.\*;    public class TreeSetDemo\_108  {  public static void main(String[] args)  {  TreeSet ts = new TreeSet();  ts.add(31); ts.add(10); ts.add(3); ts.add(21);    System.out.println("\n\t Elements = " + ts); }  } |

Program N0.

108

Therefore, besides the uniqueness of elements that a **Set** guarantees, consider using **HashSet** when ordering does not matter; using **LinkedHashSet** when you want to order elements by their insertion order; using **TreeSet** when you want to order elements by their values.

# 1. Creating a new Set



Always use generics to declare a **Set** of specific type, e.g. a **Set** of integer numbers:

**Set<Integer> numbers = new HashSet<>();**

Remember using the interface type (Set) on as the reference type, and concrete implementation (**HashSet**, **LinkedHashSet**, **TreeSet**, etc) as the actual object type:

**Set<String> names = new LinkedHashSet<>();**

We can create a Set from an existing collection. This is a trick to remove duplicate elements in non-Set collection. Consider the following code snippet:

**List<Integer> listNumbers = Arrays.asList(3, 9, 1, 4, 7, 2, 5, 3, 8, 9, 1, 3, 8, 6);**

**System.out.println(listNumbers);**

**Set<Integer> uniqueNumbers = new HashSet<>(listNumbers);**

**System.out.println(uniqueNumbers);**

**Output:**

**[3, 9, 1, 4, 7, 2, 5, 3, 8, 9, 1, 3, 8, 6]**

**[1, 2, 3, 4, 5, 6, 7, 8, 9]**

You see, the list listNumbers contains duplicate numbers, and the set uniqueNumbers removes the duplicate ones.

As with Java 8, we cand use stream with filter and collection functions to return a Set from a collection. The following code collects only odd numbers to a Set from the listNumbers above:

Set<Integer> uniqueOddNumbers = listNumbers.stream().filter(number -> number

% 2 != 0).collect(Collectors.toSet());

System.out.println(uniqueOddNumbers);

Output:

[1, 3, 5, 7, 9]

Note that the default, initial capacity of a HashSet and LinkedHashSet is **16**, so if you are sure that your Set contains more than 16 elements, it’s better to specify a capacity in the constructor. For example:

Set<String> bigNames = new HashSet<>(1000);

This creates a new HashSet with initial capacity is 1000 elements.

# 2. Performing Basic Operations on a Set



Adding elements to a Set:

The **add()** method returns true if the set does not contain the specified element, and returns false if the set already contains the specified element:

Set<String> names = new HashSet<>(); names.add("Tom"); names.add("Mary"); if (names.add("Peter")) {

System.out.println("Peter is added to the set");

}

if (!names.add("Tom")) {

System.out.println("Tom is already added to the set"); }

Output:

Peter is added to the set

Tom is already added to the set

**The Set can contain a null element:**

names.add(null);

Removing an element from a Set:

The **remove(object)** method removes the specified element from the set if it is present (the method returns true, or false otherwise) :

if (names.remove("Mary")) {

System.out.println("Marry is removed"); }

Note that the objects in the Set should implement the equals() and hashCode() methods correctly so the Set can find and remove the objects.

Check if a Set is empty:

The **isEmpty()** method returns true if the set contains no elements, otherwise returns false:

if (names.isEmpty()) {

System.out.println("The set is empty");

} else {

System.out.println("The set is not empty"); }

Remove all elements from a Set:

The **clear()** method removes all elements from the set. The set will be empty afterward:

names.clear(); if (names.isEmpty()) {

System.out.println("The set is empty"); }

Get total number of elements in a Set:

The **size()** method returns the number of elements contained in the set:

Set<String> names = new HashSet<>(); names.add("Tom"); names.add("Mary"); names.add("Peter"); names.add("Alice");

System.out.printf("The set has %d elements", names.size());

Output:

The set has 4 elements

Note that the Set interface does not provide any API for retrieving a specific element due to its nature of unordered. Except the TreeSet implementation allows retrieving the first and the last elements.

# 3. Iterating over elements in a Set



Using an iterator:

Set<String> names = new HashSet<>(); names.add("Tom"); names.add("Mary"); names.add("Peter"); names.add("Alice");

Iterator<String> iterator = names.iterator(); while (iterator.hasNext()) {

String name = iterator.next();

System.out.println(name);

}

Output:

Tom

Alice

Peter

Mary

Using the enhanced for loop:

for (String name : names) {

System.out.println(name);

}

Using the **forEach()** method with Lambda expression in Java 8:

names.forEach(System.out::println);

# 4. Searching for an element in a Set



The **contains(Object)** method returns true if the set contains the specified element, or return false otherwise. For example:

Set<String> names = new HashSet<>(); names.add("Tom"); names.add("Mary"); names.add("Peter"); names.add("Alice"); if (names.contains("Mary")) {

System.out.println("Found Mary"); }

Note that if the set contains custom objects of your own type, e.g. Student or Employee, the object should implement the equals() and hashCode() methods correctly so the Set can find the objects.

# 5. Performing Bulk Operations between two Sets



We can perform some mathematic-like operations between two sets such as subset, union, intersection and set difference. Suppose that we have two sets s1 and s2.

Subset operation: **s1.containsAll(s2)** returns true if s2 is a subset of s1 (s2 is a subset of s1 if s1 contains all of the elements in s2).

Example:

Set<Integer> s1 = new HashSet<>(Arrays.asList(20, 56, 89, 31, 8, 5));

Set<Integer> s2 = new HashSet<>(Arrays.asList(8, 89));

if (s1.containsAll(s2)) {

System.out.println("s2 is a subset of s1"); }

Output:

s2 is a subset of s1

Union operation:

s1.addAll(s2) transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.) Example:

Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 3, 5, 7, 9));

Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8)); System.out.println("s1 before union: " + s1); s1.addAll(s2);

System.out.println("s1 after union: " + s1);

Output:

s1 before union: [1, 3, 5, 7, 9] s1 after union: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Intersection operation:

s1.retainAll(s2) - transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.) Example:

Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));

Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8)); System.out.println("s1 before intersection: " + s1); s1.retainAll(s2);

System.out.println("s1 after intersection: " + s1);

Output:

s1 before intersection: [1, 2, 3, 4, 5, 7, 9] s1 after intersection: [2, 4]

Set difference operation:

s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.) Example:

Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));

Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8)); System.out.println("s1 before difference: " + s1); s1.removeAll(s2);

System.out.println("s1 after difference: " + s1);

Output:

s1 before difference: [1, 2, 3, 4, 5, 7, 9] s1 after difference: [1, 3, 5, 7, 9]

# 6. Concurrent Sets



All three implementations HashSet, LinkedHashSet and TreeSet are not synchronized. So if you use them in concurrent context (multi-threads), you have to synchronize them externally using **Collections.synchronizedSet()** static method. For example:

Set<Integer> numbers = Collections.synchronizedSet(new HashSet<Integer>());

The returned set is synchronized (thread-safe). And remember you must manually synchronize on the returned set when iterating over it:

synchronized (numbers) {

Iterator<Integer> iterator = numbers.iterator(); while (iterator.hasNext()) {

|  | Integer number = iterator.next(); |
| --- | --- |
| } | System.out.println(number);  } |
|  |  |

# Summary of Set Collection



| **Property** | **HashSet** | **LInkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| Ordered | Unordered | Ordered by insertion | Sorted order |
| Null Values | Allowed | Allowed | Allowed |
| Duplicate | Not Allowed | Not Allowed | Not Allowed |
| Synchronized | No | No | No |
| Initial Capacity | 16 | 16 | Not Applicable |
| Data Structure | HashTable | HashTable + Double Linked List | Balanced Tree |

# Equals() and hashCode()



When it comes to working with collections, we should override the equals()and hashCode() methods properly in the classes of the elements being added to the collections. Otherwise we will get unexpected behaviors or undesired results.

You know, the Object class (the super class of all classes in Java) defines two methods equals() and hashCode(). That means all classes in Java (including the ones you created) inherit these methods. Basically, the Object class implements these methods for general purpose so you may not see them frequently.

However, you will have to override them specifically for the classes whose objects are added to collections, especially the hashtable-based collections such as HashSet and HashMap.

Understanding the equals() method:



When comparing two objects together, Java calls their equals() method which returns true if the two objects are equal, or false otherwise. Note that this comparison using equals() method is very different than using the == operator.

Here’s the difference:

The equals() method is designed to compare two objects semantically (by comparing the data members of the class), whereas the == operator compares two objects technically (by comparing their references i.e. memory addresses).

**NOTE:** The implementation of equals() method in the Object class compares references of two objects. That means we should override it in our classes for semantic comparison. Almost classes in the JDK override their own version of equals() method, such as String, Date, Integer, Double, etc.

A typical example is String comparison in Java. Let’s see the following code:

String s1 = new String("This is a string"); String s2 = new String("This is a string"); boolean refEqual = (s1 == s2); boolean secEqual = (s1.equals(s2));

System.out.println("s1 == s2: " + refEqual);

System.out.println("s1.equals(s2): " + secEqual);

Output:

s1 == s2: false s1.equals(s2): true

You see, the reference comparison (== operator) returns false because s1 and s2are two different objects which are stored in different locations in memory. Whereas the semantic comparison returns true because s1 and s2 has same value (“This is a string”) which can be considered equal semantically.

Likewise, let say we have the Student class as following:

public class Student { private String id; private String name; private String email; private int age;

public Student(String id, String name, String email, int age) {  [this.id](http://this.id/) = id;  [this.name](http://this.name/) = name; this.email = email; this.age = age;

}

public String toString() {

String studentInfo = "Student " + id; studentInfo += ": " + name; studentInfo += " - " + email; studentInfo += " - " + age;

return studentInfo;

}

}

In practice, we can consider two Student objects are semantically equal if they have same attributes (id, name, email and age). Now, let’s see how to override the equals() method in this class to confirm that two Student objects having identical attributes are considered to be equal:

public boolean equals(Object obj) { if (obj instanceof Student) { Student another = (Student) obj; if (this.id.equals([another.id)](http://another.id/) && this.name.equals([another.name)](http://another.name/) && this.email.equals(another.email) && this.age == another.age) { return true;

}

}

return false;

}

Here, this equals()method checks if the passed object is of type Student and if it has same attributes as the current object, they are considered to be equal (return true); otherwise they are not equal (return false). Let’s test it out with the following code:

Student student1 = new Student("123", "Tom", "tom@gmail.com", 30);

Student student2 = new Student("123", "Tom", "tom@gmail.com", 30);

Student student3 = new Student("456", "Peter", "peter@gmail.com", 23);

System.out.println("student1 == student2: " + (student1 == student2));

System.out.println("student1.equals(student2): " + (student1.equals(student2)));

System.out.println("student2.equals(student3): " + (student2.equals(student3)));

And we have the following output:

student1 == student2: false student1.equals(student2): true student2.equals(student3): false

Let’s see another example to understand how overriding the equals() method really helps. Suppose that we have a list of students like this:

List<Student> listStudents = new ArrayList<>(); This list contains the three Student objects above:

listStudents.add(student1); listStudents.add(student2); listStudents.add(student3);

Now we want to check whether the list contains a student with a given ID. I’ll tell you an easy and interesting solution using equals() method.

Note that the List interface provides the contains(Object) method which can be used for checking if the specified object exists in the list. Behind the scene, the list invokes the equals() method on the search object to compare it with other objects in the collection.

And do you agree that two Student objects can be considered to be equal if they have same ID? So we update the equals() method in the Student class like this:

public boolean equals(Object obj) { if (obj instanceof Student) {

Student another = (Student) obj;

if (this.id.equals([another.id)](http://another.id/)) { return true;

}

}

return false;

}

Here, this equals() method compares only the id attribute of two Student objects. And add another constructor to the Student class:

public Student(String id) {  [this.id](http://this.id/) = id;

}

Now, we can perform the checking like this:

Student searchStudent1 = new Student("123"); Student searchStudent4 = new Student("789"); boolean found1 = listStudents.contains(searchStudent1); boolean found4 = listStudents.contains(searchStudent4);

System.out.println("Found student1: " + found1);

System.out.println("Found student4: " + found4);

Here’s the result:

Found student1: true

Found student4: false

It’s awesome, isn’t it? Thanks to the equals() method which makes our code simple. Imagine if we don’t use it, we would have implemented the search functionality more complex like this:

public boolean searchStudent(List<Student> listStudents, String id) { for (Student student : listStudents) { if (student.getId().equals(id)) { return true;

}

} return false;

}

Understanding the hashCode() method:



The Object class defines the hashCode() method as follows:

public int hashCode()

You can see this method returns an integer number. So where is it used?

Here’s the secret:

This hash number is used by hashtable-based collections like Hashtable, HashSet and HashMap to store objects in small containers called “buckets”. Each bucket is associated with a hash code, and each bucket contains only objects having identical hash code.

In other words, a hashtable groups its elements by their hash code values. This arrangement helps the hashtable locates an element quickly and efficiently by searching on small parts of the collection instead the whole collection.

Here are the steps to locate an element in a hashtable:

* Get hash code value of the specified element. This results in the hashCode() method to be invoked.
* Find the right bucket associated with that hash code.
* Inside the bucket, find the correct element by comparing the specified element with all the elements in the bucket. This results in the equals() method of the specified element to be invoked.

Having said that, when we add objects of a class to a hashtable-based collection (HashSet, HashMap), the class’s hashCode() method is invoked to produce an integer number (which can be an arbitrary value). This number is used by the collection to store and locate the objects quickly and efficiently, as a hashtable-based collection does not maintain order of its elements.

**NOTE:** The default implementation of hashCode() in the Object class returns an integer number which is the memory address of the object. We should override it in our own classes. Almost classes in the JDK override their own version of hashCode() method, such as String, Date, Integer, Double, etc.

The Rules Between equals() and hashCode():



As explained above, as hashtable-based collection locates an element by invoking

its hashCode() and equals() methods, so we must obey this contract with regard to the way we override these methods:

* When the equals() method is overridden, the hashCode() method must be overridden as well.
* If two objects are equal, their hash codes must be equal as well.
* If two objects are not equal, there’s no constraint on their hash codes (their hash codes can be equal or not).
* If two objects have identical hash codes, there’s no constraint on their equality (they can be equal or not). ▪ If two objects have different hash codes, they must not be equal.

By following these rules, we keep the collections consistent in maintaining its elements. If we violate these rules, the collections will behave unexpectedly such as the objects cannot be found, or wrong objects are returned instead of the correct ones.

Now, let’s see how the hashCode() and equals() methods affect the behaviors of a Set by coming back to the student example.

Until now, we have the Student class written like this:

public class Student { private String id; private String name; private String email; private int age;

public Student(String id) {  [this.id](http://this.id/) = id;

}

public Student(String id, String name, String email, int age) {

[this.id](http://this.id/) = id;  [this.name](http://this.name/) = name; this.email = email; this.age = age;

}

public String toString() {

String studentInfo = "Student " + id; studentInfo += ": " + name; studentInfo += " - " + email; studentInfo += " - " + age;

return studentInfo;

}

public boolean equals(Object obj) { if (obj instanceof Student) {

Student another = (Student) obj; if (this.id.equals([another.id)](http://another.id/)) { return true;

}

}

return false;

}

}

Note that, there’s only equals() method is overridden till now.

We add three Student objects to a HashSet as shown in the following code:

Student student1 = new Student("123", "Tom", "tom@gmail.com", 30);

Student student2 = new Student("123", "Tom", "tom@gmail.com", 30);

Student student3 = new Student("456", "Peter", "peter@gmail.com", 23); Set<Student> setStudents = new HashSet<Student>(); setStudents.add(student1); setStudents.add(student2); setStudents.add(student3);

Now, let’s print information of all students in this set using Lambda expressions:

setStudents.forEach(student -> System.out.println(student));

And we have the following output:

Student 456: Peter - peter@gmail.com - 23

Student 123: Tom - tom@gmail.com - 30

Student 123: Tom - tom@gmail.com - 30

Look, do you notice that there seems to be 2 duplicate students (ID: 123), right?

Oh, we would expect the set does not contain duplicate elements, why is this possible?

Here’s the reason:

The set invokes the equals() and hashCode() methods on each object being added to ensure there’s no duplication. In our case, the Student class overrides only the equals() method. And the hashCode() method inherited from the Object class returns memory addresses of each object which is not consistent with

the equals() method (the contact is violated). Therefore the set treats the student1 and student2 object as two different elements.

Now, let’s override the hashCode() method in the Student class to obey the contract of equals() and hashCode(). Here’s the code needs to be added:

public int hashCode() {

return 31 + id.hashCode(); }

This method returns an integer number based on the hash code of the id attribute (its hashCode() method is overridden by the String class). Run the code to print the set again and observe the result:

Student 123: Tom - tom@gmail.com - 30

Student 456: Peter - peter@gmail.com - 23

Awesome! The duplicate element is now removed, you see? That’s exactly what we want.

With the equals() and hashCode() methods overridden properly, we can also perform search on the set like this: Student searchStudent = new Student("456"); boolean found = setStudents.contains(searchStudent);

System.out.println("Found student: " + found);

Output:

Found student: true

For more experiments yourself, try to remove either the equals() or hashCode() method and observe the outcome.

Vishal Shah, I hope the above explanation and examples help you understand how the equals() and hashCode() methods work, and why they play important roles with regard to collections.

There’s more tips about implementing equals() and hashCode() methods correctly and efficiently, which you can find in the following article: <http://www.javaranch.com/journal/2002/10/equalhash.html>

# Sorting Collection



To understand object ordering properly, let’s see some examples where we use the utility class Collections to sort elements of a collection (or Arrays class to sort elements in an array):

* Collections.sort(list): sorts a List collection.
* Arrays.sort(array): sorts an array.

## Example #1: Sorting a list of String objects

List<String> names = Arrays.asList(

"Tom", "Peter", "Alice", "Bob", "Sam",

"Mary", "Jane", "Bill", "Tim", "Kevin");

System.out.println("Before sorting: " + names);

Collections.sort(names);

System.out.println("After sorting: " + names);

Output:

Before sorting: [Tom, Peter, Alice, Bob, Sam, Mary, Jane, Bill, Tim, Kevin]

After sorting: [Alice, Bill, Bob, Jane, Kevin, Mary, Peter, Sam, Tim, Tom]

In this example, the list names is sorted by alphabetic order of String.

## Example #2: Sorting a list of Integer objects

List<Integer> numbers = Arrays.asList(8, 2, 5, 1, 3, 4, 9, 6, 7, 10);

System.out.println("Before sorting: " + numbers);

Collections.sort(numbers);

System.out.println("After sorting: " + numbers);

Output:

Before sorting: [8, 2, 5, 1, 3, 4, 9, 6, 7, 10]

After sorting: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Here, the integer numbers in the list numbers are sorted by alphanumeric order.

## Example #3: Sorting a list of Date objects

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

List<Date> birthdays = new ArrayList<>(); birthdays.add(dateFormat.parse("2016-01-20")); birthdays.add(dateFormat.parse("1998-12-03")); birthdays.add(dateFormat.parse("2009-07-15")); birthdays.add(dateFormat.parse("2012-04-30")); System.out.println("Before sorting: "); for (Date date : birthdays) {

System.out.println(dateFormat.format(date));

}

Collections.sort(birthdays);

System.out.println("After sorting: ");

for (Date date : birthdays) {

System.out.println(dateFormat.format(date)); }

Output:

Before sorting:

2016-01-20

1998-12-03

2009-07-15

2012-04-30

After sorting:

1998-12-03

2009-07-15

2012-04-30

2016-01-20

Here, the list birthdays is sorted by chronological order of its elements - objects of type Date.

From the 3 examples above, the collections are sorted by natural ordering of its elements:

* The natural ordering of String objects is alphabetic order.
* The natural ordering of Integer objects is alphanumeric order.
* The natural ordering of Date objects is chronological order.

# Understanding Natural Ordering



**Natural ordering** is the default ordering of objects of a specific type when they are sorted in an array or a collection. The Java language provides the **Comparable** interface that allows us define the natural ordering of a class. This interface is declared as follows:

public interface Comparable<T> { public int compareTo(T object); }

As you can see, this interface is parameterized (generics) and it has a single method compareTo() that allows two objects of a same type to be compared with each other. The important point here is the value returned by this method: an integer number indicates the comparison result of two objects. Remember these rules:

* Compare value = 0: two objects are equal.
* Compare value > 0: the first object (the current object) is greater than the second one. ▪ Compare value < 0: the first object is less than the second one.

Imagine that, when the objects are being sorted, their compareTo() methods are invoked to compare with other objects. And based on the compare value returned, the objects are sorted by natural ordering.

Classes whose objects used in collections or arrays should implement the Comparable interface for providing the natural ordering of its objects when being sorted. Otherwise we will get an error at runtime.

A class that implements the Comparable interface is said to have **class natural ordering.** And the compareTo() method is called the **natural comparison method**.

In the above examples, we don’t have to write code to implement the Comparable interface because the String, Integer and Date classes already implemented this interface. Hence we can sort a collection containing objects of these types.

Other wrapper types in Java are also comparable: Long, Double, Float, etc.

When we create our own type, we have to implement the Comparable interface in order to have objects of our type eligible to be sorted in collections or arrays. Let’s see an example to understand how the Comparable interface is used.

Let’s say we have the Employee class which is defined as shown below:

public class Employee { String firstName;

String lastName;

Date joinDate;

public Employee(String firstName, String lastName) { this.firstName = firstName; this.lastName = lastName;

}

public String toString() {

return firstName + " " + lastName;

}

// getters and setters

}

Add some employees to a list collection like this:

List<Employee> listEmployees = new ArrayList<>();

Employee employee1 = new Employee("Tom", "Eagar");

Employee employee2 = new Employee("Tom", "Smith");

Employee employee3 = new Employee("Bill", "Joy");

Employee employee4 = new Employee("Bill", "Gates");

Employee employee5 = new Employee("Alice", "Wooden");

listEmployees.add(employee1); listEmployees.add(employee2); listEmployees.add(employee3); listEmployees.add(employee4); listEmployees.add(employee5); Try to sort this list:

Collections.sort(listEmployees);

We will get an error at runtime: **no suitable method found for sort(List<Emmployee>)…**

WHY?

It’s because the Employee class doesn’t implement the Comparable interface so the sort() method cannot compare the objects.

Now, let’s have the Employee class implements the Comparable interface, and we define the natural ordering is first name - last name, meaning the employees are sorted by first name first, then by last name. Here’s the updated version of the Employee class:

public class Employee implements Comparable<Employee> { // fields...

// constructors...

// getters...

// setters...

// implement the natural comparison method:

public int compareTo(Employee another) {

int compareValue = this.firstName.compareTo(another.firstName); if (compareValue == 0) {

return this.lastName.compareTo(another.lastName);

}

return compareValue; }

}

Look at how the compareTo() method is implemented here:

* First, we compare the first name by using the String’s compareTo() method. We can safely use this method of the built-in types in Java: String, Date, Integer, Long, etc.
* If two employees have same first name (compare value = 0), then we compare their last name. Finally the compare value is returned as per the contract of the Comparable interface.

Now, run this test code and observe the result:

System.out.println("Before sorting: " + listEmployees);

Collections.sort(listEmployees);

System.out.println("After sorting: " + listEmployees);

Output:

Before sorting: [Tom Eagar, Tom Smith, Bill Joy, Bill Gates, Alice Wooden]

After sorting: [Alice Wooden, Bill Gates, Bill Joy, Tom Eagar, Tom Smith]

Awesome! It works perfectly as we expected: the employees are sorted by their first name, and then last name.

Note #1:

We cannot compare objects of different types, e.g. a String object cannot be compared with an Integer object. As the compareTo() method enforces this rule, we can only compare objects of the same type. If we add objects of different types to a collection and sort it, we will get ClassCastException.

Note #2:

If we want to reverse the natural ordering, simply swap the objects being compared in the compareTo() method. For example, the following implementation sorts employees by their first name into descending order:

public int compareTo(Employee another) {

return another.firstName.compareTo(this.firstName); }

In case we use a sorted collection i.e. TreeSet, we don’t have to use the Collections.sort() utility method, as a TreeSet sorts its elements by their natural ordering. The following example demonstrates how to use a TreeSet to sort Strings:

Set<String> setNames = new TreeSet<>();

setNames.addAll(Arrays.asList("Tom", "Peter", "Alice", "Bob", "Sam",

"Mary", "Jane", "Bill", "Tim", "Kevin")); System.out.println(setNames);

Output:

[Alice, Bill, Bob, Jane, Kevin, Mary, Peter, Sam, Tim, Tom]

Similarly, we can sort the Employee objects using a TreeSet like this:

Set<Employee> setEmployees = new TreeSet<>();

Employee employee1 = new Employee("Tom", "Eagar");

Employee employee2 = new Employee("Tom", "Smith");

Employee employee3 = new Employee("Bill", "Joy");

Employee employee4 = new Employee("Bill", "Gates");

Employee employee5 = new Employee("Alice", "Wooden");

setEmployees.add(employee1); setEmployees.add(employee2); setEmployees.add(employee3); setEmployees.add(employee4); setEmployees.add(employee5);

System.out.println(setEmployees);

Output:

[Alice Wooden, Bill Gates, Bill Joy, Tom Eagar, Tom Smith]

So far we have got understanding about the natural ordering of objects and how the Comparable interface defines the ordering.

What if we want to sort objects in an order which differs from the natural ordering? For example, sort the employees list above by seniority (based on their join dates)?

# Understanding Comparator



The Collections utility class provides a method for sorting a list using an external comparator:

**Collections.sort(list, comparator)**

This overloaded version takes two parameters: a list collection and a comparator, which is any object that implements the Comparator interface. This interface declares this method:

public interface Comparator<T> { public int compare(T obj1, T obj2); }

Like the Comparable interface, this interface is also parameterized for any specific type. The compare() method is similar except it takes both the objects to be compared as arguments. The return value is also evaluated similarly.

For example, the following class compares two Employee objects using the Comparator interface:

public class EmployeeComparator implements Comparator<Employee> { public int compare(Employee emp1, Employee emp2) {

return emp1.getJoinDate().compareTo(emp2.getJoinDate());

}

}

In this comparator, we compare two Employee objects by their join dates. And update the Employee class like this (add an overloaded constructor and update the toString() method):

public class Employee implements Comparable<Employee> { // fields...

// getters & setters....

// constructor public Employee(String firstName, String lastName, Date joinDate) { this.firstName = firstName; this.lastName = lastName; this.joinDate = joinDate;

}

public String toString() {

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

return firstName + " " + lastName + " " + dateFormat.format(joinDate);

}

}

And here’s the test code:

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

List<Employee> listEmployees = new ArrayList<>();

Employee employee1 = new Employee("Tom", "Eagar", dateFormat.parse("2007-12-03"));

Employee employee2 = new Employee("Tom", "Smith", dateFormat.parse("2005-06-20"));

Employee employee3 = new Employee("Bill", "Joy", dateFormat.parse("2009-01-31"));

Employee employee4 = new Employee("Bill", "Gates", dateFormat.parse("2005-05-12"));

Employee employee5 = new Employee("Alice", "Wooden", dateFormat.parse("2009-01-

22"));

listEmployees.add(employee1); listEmployees.add(employee2); listEmployees.add(employee3); listEmployees.add(employee4); listEmployees.add(employee5);

System.out.println("Before sorting: ");

System.out.println(listEmployees);

Collections.sort(listEmployees, new EmployeeComparator());

System.out.println("After sorting: ");

System.out.println(listEmployees);

Collections.sort(listEmployees, (emp1, emp2) -> emp1.getJoinDate().compareTo(emp2.getJoinDate()));

Output:

Before sorting:

[Tom Eagar 2007-12-03, Tom Smith 2005-06-20, Bill Joy 2009-01-31, Bill Gates 2005-

05-12, Alice Wooden 2009-01-22]

After sorting:

[Bill Gates 2005-05-12, Tom Smith 2005-06-20, Tom Eagar 2007-12-03, Alice Wooden 2009-01-22, Bill Joy 2009-01-31]

Note #3:

Since Java 8, we can use Lambda expressions to create a comparator more easily like this:

Collections.sort(listEmployees,

(emp1, emp2) -> emp1.getJoinDate().compareTo(emp2.getJoinDate()));

We can also pass a comparator when creating a new instance of a TreeSet like this:

Set<Employee> setEmployees = new TreeSet<>(new EmployeeComparator());

Then the TreeSet will sort its elements according to the order defined by the specified comparator.

Using a comparator is useful in the following scenarios:

* The class doesn’t have natural ordering (or we don’t have source code to update it).
* We want to sort objects in orders other than the natural ordering.
* We want to provide multiple ways for sorting the objects, e.g. one comparator for each sorting criteria.

# The constraint between natural ordering and equals



Now, let’s discuss about the constraint between natural ordering and equals() method.

You know, the documentation of both Comparable and Comparator states that the natural ordering and the ordering specified by a comparator should be consistent with the equals() method of the class. Let’s say we have two objects obj1 and obj2 of class A, then:

If obj1.compareTo(obj2) = 0 then obj1.equals(obj2) = true

If this contract is violated, we will get strange behavior when using sorted collections such as TreeSet and TreeMap.

Let’s examine an example to understand why this constraint really matters. Come back to the example of sorting a list of Employee objects I provided in the previous email.

We haven’t overridden the equals() method yet. Now, let’s override it for the Employee class:

public class Employee implements Comparable<Employee> {

// fields, constructors, getters and setters and toString()... public int compareTo(Employee another) { int compareValue = this.firstName.compareTo(another.firstName); if (compareValue == 0) { return this.lastName.compareTo(another.lastName);

}

return compareValue;

} public boolean equals(Object obj) { if (obj instanceof Employee) {

Employee another = (Employee) obj; if (this.firstName.equals(another.firstName)

&& this.lastName.equals(another.lastName))

{

return true;

}

}

return false;

} }

Currently, it is compatible with the compareTo() method which also compares first name and then last name.

What if we need to change the compareTo() method for comparing two Employee objects by their seniority (join date) like this:

public int compareTo(Employee another) {

return this.joinDate.compareTo(another.joinDate); }

Let’s execute some test code to see the outcome:

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

Set<Employee2> setEmployees = new TreeSet<>(new EmployeeComparator2());

Employee2 employee1 = new Employee2("Tom", "Eagar", dateFormat.parse("2007-1203"));

Employee2 employee2 = new Employee2("Tom", "Smith", dateFormat.parse("2005-0620"));

Employee2 employee3 = new Employee2("Bill", "Joy", dateFormat.parse("2007-12-03"));

Employee2 employee4 = new Employee2("Bill", "Gates", dateFormat.parse("2005-0512"));

Employee2 employee5 = new Employee2("Alice", "Wooden", dateFormat.parse("2005-06-

20"));

setEmployees.add(employee1); setEmployees.add(employee2); setEmployees.add(employee3); setEmployees.add(employee4); setEmployees.add(employee5);

System.out.println(setEmployees);

Note that the employee1 and employe5 have same join date, so do the employee3 and employee4. Add all of these 5 objects to the set:

setEmployees.add(employee1); setEmployees.add(employee2); setEmployees.add(employee3); setEmployees.add(employee4); setEmployees.add(employee5);

And print the set:

System.out.println(setEmployees);

Can you guess the output? Here is it:

[Tom Smith 2005-06-20, Tom Eagar 2007-12-03, Bill Joy 2009-01-31]

Ouch! Why are there only 3 employees in the set?

It’s because the set compares the objects using the compareTo() method which considers two employees are equal if they have same join date, whereas the set does not allow duplicate elements, hence the employee4 and employee5 objects are not added to the set.

Now, you understand the consequence if natural ordering and equals are not consistent, right?

So is there any solution or workaround?

Suppose that we still want to keep the natural ordering based on join date, while keep compatible with the equals() method, here’s how we update the compareTo() method:

That’s it! In this solution, we compare the Employee objects by their join dates first. If equal, continue comparing by their first names. And if equal, continue comparing their last names. This way we can keep the compareTo() method compatible with the equals() method.

Run the test code again and observe the output:

[Tom Smith 2005-06-20, Alice Wooden 2007-12-03, Tom Eagar 2007-12-03, Bill Gates 2009-01-31, Bill Joy 2009-01-31]

The same problem and solution applies for a comparator.

What is a Map?



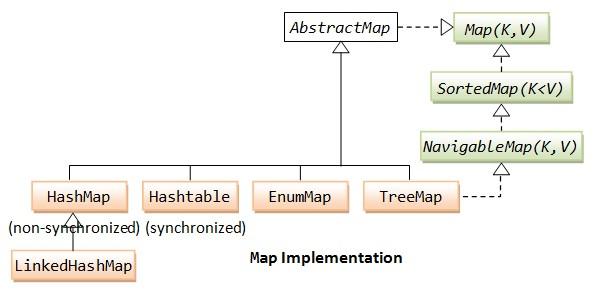
A map is a collection of key-value pairs (e.g., name-address, name-phone, isbn-title, word-count). Each key maps to one and only value. Duplicate keys are not allowed, but duplicate values are allowed. Maps are similar to linear arrays, except that an array uses an integer key to index and access its elements; whereas a map uses any arbitrary key (such as Strings or any objects).

Note that a Map is not considered to be a true collection, as the Map interface does not extend the Collection interface. Instead, it starts an independent branch in the Java Collections Framework

The interface Map<K,V>, which takes two generic types K and V and read as Map of Key type K and Value type V, is used as a collection of "key-value pairs". No duplicate key is allowed. Frequently-used implementations

include HashMap, Hashtable and LinkedHashMap. Its sub-interface SortedMap<K, V> models an ordered and sorted map, based on its key, implemented in TreeMap.

Take note that Map<K,V> is not a sub-interface of Collection<E>, as it involves a pair of objects for each element



# Characteristics of a Map



Because a Map is not a true collection, its characteristics and behaviors are different than the other collections like List or Set.

A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value (HashMap and LinkedHashMap) but some does not (TreeMap).

The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.

Why and When Use Maps:



Maps are perfectly for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform lookups by keys. A couple of examples:

* A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
* A map of classes and students. Each class (key) is associated with a list of students (value).

Implementations of Map:



In the inheritance tree of the Map interface, there are several implementations but only 3 major, common, and general purpose implementations - they are HashMap and LinkedHashMap and TreeMap.

## HashMap

This implementation uses a hash table as the underlying data structure. It implements all of the Map operations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its keyvalue elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable.

## LinkedHashMap

This implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order.

## TreeMap

This implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify. The Map<K,V> interface declares the following abstract methods:

| **Method** | **Description** |
| --- | --- |
| V get(Object key) | Returns the value of the specified key |
| V put(K key, V value) | Associate the specified value with the specified key |
| boolean containsKey(Object key) | Searches for the specified key |
| boolean containsValue(Object value) | Searches for the specified value |
| Set<K> keySet() | Returns a set view of the keys |
| Collection<V> values() | Returns a collection view of the values |
| Set entrySet() | Returns a set view of the key-value |

# 1. Creating a new Map



Creating a HashMap:

Always use interface type (Map), generics and diamond operator to declare a new map. The following code creates a HashMap:

Map<Integer, String> mapHttpErrors = new HashMap<>(); mapHttpErrors.put(200, "OK"); mapHttpErrors.put(303, "See Other"); mapHttpErrors.put(404, "Not Found"); mapHttpErrors.put(500, "Internal Server Error");

System.out.println(mapHttpErrors);

This maps HTTP status codes to their descriptions. Output:

{404=Not Found, 500=Internal Server Error, 200=OK, 303=See Other}

As you can see in the output, a HashMap does not impose any order on its key-value elements.

You can create a new Map that copies elements from an existing map. For example:

Map<Integer, String> mapErrors = new HashMap<>(mapHttpErrors); The map mapErrors is created with initial elements copied from the map mapHttpErrors.

Creating a LinkedHashMap:

The following code creates a LinkedHashMap that maps phone numbers with contact names: Map<String, String> mapContacts = new LinkedHashMap<>(); mapContacts.put("0169238175", "Tom"); mapContacts.put("0904891321", "Peter"); mapContacts.put("0945678912", "Mary"); mapContacts.put("0981127421", "John");

System.out.println(mapContacts); Output:

{0169238175=Tom, 0904891321=Peter, 0945678912=Mary, 0981127421=John} As you can see, the LinkedHashMap maintains its elements by their insertion order.

Creating a TreeMap:

The following code creates a TreeMap that maps file extensions to programming languages:

Map<String, String> mapLang = new TreeMap<>(); mapLang.put(".c", "C"); mapLang.put(".java", "Java"); mapLang.put(".pl", "Perl");

mapLang.put(".cs", "C#"); mapLang.put(".php", "PHP"); mapLang.put(".cpp", "C++"); mapLang.put(".xml", "XML");

System.out.println(mapLang);

Output:

{.c=C, .cpp=C++, .cs=C#, .java=Java, .php=PHP, .pl=Perl, .xml=XML}

As you can see, the TreeMap sorts its keys by their natural ordering, which is the alphabetical order in this case.

# 2. Performing Basic Operations on a Map



The basic operations of a Map are association (put), lookup (get), checking (containsKey and containsValue), modification (remove and replace) and cardinality (size and isEmpty).

Associating a value with a key:

The put(K, V) method associates the specified value V with the specified key K. If the map already contains a mapping for the key, the old value is replaced by the specified value:

Map<Integer, String> mapHttpErrors = new HashMap<>(); mapHttpErrors.put(400, "Bad Request"); mapHttpErrors.put(304, "Not Modified"); mapHttpErrors.put(200, "OK"); mapHttpErrors.put(301, "Moved Permanently"); mapHttpErrors.put(500, "Internal Server Error");

Getting a value associated with a specified key:

The get(Object key) method returns the value associated with the specified key, or returns null if the map contains no mapping for the key. Given the map in the previous example:

String status301 = mapHttpErrors.get(301);

System.out.println("301: " + status301);

Output:

301: Moved Permanently

Checking if the map contains a specified key:

The method containsKey(Object key) returns true if the map contains a mapping for the specified key. For example:

if (mapHttpErrors.containsKey("200")) {

System.out.println("Http status 200");

}

Output:

Found: Http status 200

Checking if the map contains a specified value:

The method containsValue(Object value) returns true if the map contains one or more keys associated with the specified value. For example:

if (mapHttpErrors.containsValue("OK")) {

System.out.println("Found status OK"); }

Output:

Found status OK

Removing a mapping from the map:

The remove(Object key) method removes the mapping for a key from the map if it is present (we care about only the key, and the value does not matter). This method returns the value to which the map previously associated the key, or null if the map doesn’t contain mapping for the key. Here’s an example: String removedValue = mapHttpErrors.remove(500); if (removedValue != null) {

System.out.println("Removed value: " + removedValue); } Output:

Removed value: Internal Server Error

Similarly, the remove(Object key, Object value) method removes the mapping of a specified key and specified value, and returns true if the value was removed. This method is useful in case we really care about the key and value to be removed.

Replacing a value associated with a specified key:

The replace(K key, V value) method replaces the entry for the specified key only if it is currently mapping to some value. This method returns the previous value associated with the specified key. Here’s an example:

System.out.println("Map before: " + mapHttpErrors); mapHttpErrors.replace(304, "No Changes");

System.out.println("Map after: " + mapHttpErrors);

Output:

Map before: {400=Bad Request, 304=Not Modified, 200=OK, 301=Moved Permanently}

Map after: {400=Bad Request, 304=No Changes, 200=OK, 301=Moved Permanently}

Similarly, the replace(K key, V oldValue, V newValue) method replaces the entry for the specified key only if it is currently mapping to the specified value. This method returns true if the value was replaced. Useful in case we want to replace exactly a key-value mapping.

Getting the size of the map:

The size() method returns the number of key-value mappings in this map. For example:

int size = mapHttpErrors.size();

Output:

Number of HTTP status code: 5

Checking if the map is empty:

The isEmpty() method returns true if the map contains no key-value mappings. For example:

if (mapHttpErrors.isEmpty()) {

System.out.println("No Error");

} else {

System.out.println("Have HTTP Errors"); } Output:

Have HTTP Errors

# 3. Iterating Over a Map (using Collection views)



As a Map is not a true collection, there is no direct method for iterating over a map. Instead, we can iterate over a map using its collection views. Any Map’s implementation has to provide the following three Collection view methods: keyset()

keySet(): returns a Set view of the keys contained in the map. Hence we can iterate over the keys of the map as shown in the following example:

Map<String, String> mapCountryCodes = new HashMap<>(); mapCountryCodes.put("1", "USA"); mapCountryCodes.put("44", "United Kingdom"); mapCountryCodes.put("33", "France"); mapCountryCodes.put("81", "Japan");

Set<String> setCodes = mapCountryCodes.keySet(); Iterator<String> iterator = setCodes.iterator();

while (iterator.hasNext()) {

String code = iterator.next();

String country = mapCountryCodes.get(code);

System.out.println(code + " => " + country); }

Output:

44 => United Kingdom

33 => France

1 => USA

81 > Japan

values() values(): returns a collection of values contained in the map. Thus we can iterate over values of the map like this:

Collection<String> countries = mapCountryCodes.values(); for (String country : countries) {

System.out.println(country); } Output:

United Kingdom

France

USA

Japan

## entryset()

entrySet(): returns a Set view of the mappings contained in this map. Therefore we can iterate over mappings in the map like this:

Set<Map.Entry<String, String>> entries = mapCountryCodes.entrySet(); for (Map.Entry<String, String> entry : entries) {

String code = entry.getKey();

String country = entry.getValue();

System.out.println(code + " => " + country); } Output:

44 => United Kingdom

33 => France

1 => USA

81 => Japan

Since Java 8 with Lambda expressions and the **forEach()** statement, iterating over a Map is as easy as:

mapCountryCodes.forEach(

(code, country) -> System.out.println(code + " => " + country)); Output:

44 => United Kingdom

33 => France

1 => USA

81 => Japan

# 4. Performing Bulk Operations with Maps



There are two bulk operations with maps: clear() and putAll().

The clear() method removes all mappings from the map. The map will be empty after this method returns. For example:

mapHttpErrors.clear();

System.out.println("Is map empty? " + mapHttpErrors.isEmpty()); Output:

Is map empty? true

The putAll(Map<K, V> m) method copies all of the mappings from the specified map to this map. Here’s an example: Map<Integer, String> countryCodesEU = new HashMap<>(); countryCodesEU.put(44, "United Kingdom"); countryCodesEU.put(33, "France"); countryCodesEU.put(49, "Germany");

Map<Integer, String> countryCodesWorld = new HashMap<>(); countryCodesWorld.put(1, "United States"); countryCodesWorld.put(86, "China"); countryCodesWorld.put(82, "South Korea");

System.out.println("Before: " + countryCodesWorld); countryCodesWorld.putAll(countryCodesEU);

System.out.println("After: " + countryCodesWorld); Output:

Before: {1=United States, 82=South Korea, 86=China}

After: {1=United States, 33=France, 49=Germany, 82=South Korea, 86=China, 44=United Kingdom}

# 5. Concurrent Maps



Unlike the legacy Hashtable which is synchronized, the HashMap, TreeMap and LinkedHashMap are not synchronized.

If thread-safe is priority, consider using ConcurrentHashMap in place of HashMap. Or we can use the

Collections.synchronizedMap() utility method that returns a synchronized (thread-safe) map backed by the specified map. For example:

Map<Integer, String> map = Collections.synchronizedMap(new HashMap<>());

And remember we have to manually synchronize the map when iterating over any of its collection views: Set<Integer> keySet = map.keySet(); synchronized (map) {

Iterator<Integer> iterator = keySet.iterator(); while (iterator.hasNext()) {

Integer key = iterator.next();

String value = map.get(key);

}

}

If you use a kind of SortedMap, e.g. TreeMap, consider using the more specific method Collections.synchronizedSortedMap().

**NOTE:** If you use your own type for the key and value (e.g. Student or Employee), the key class and value class must implement the equals() and hashCode() methods properly so that the map can look up them correctly.

# Summary of Map Implementation



| **Property** | **HashMap** | **LinkedHashMap** | **TreeMap** | **HashTable** |
| --- | --- | --- | --- | --- |
| Ordered | Unordered | Ordered by insertion | Sorted Order | Unordered |
| Null Values | Yes | Yes | Allowed | No |
| Duplicate | Keys =NO  Value = yes | Keys =NO  Value = yes | Keys =NO  Value = yes | Keys =NO  Value = yes |
| Synchronized | NO | NO | NO | YES |
| Initial Capacity | 16 | 16 | Not Applicable | 11 |
| Data Structure | HashTable | HashTable + Double  Linked List | Red Black Tree | HashTable |

Queue?



Queue means ‘waiting line’, which is very similar to queues in real life: a queue of people standing in an airport’s checkin gate; a queue of cars waiting for green light in a road in the city; a queue of customers waiting to be served in a bank’s counter, etc.

In programming, queue is a data structure that holds elements prior to processing, similar to queues in real-life scenarios. Let’s consider a queue holds a list of waiting customers in a bank’s counter. Each customer is served one after another, follow the order they appear or registered.

The first customer comes is served first, and after him is the 2nd, the 3rd, and so on. When serving a customer is done, he or she lefts the counter (removed from the queue), and the next customer is picked to be served next. Other customers come later are added to the end of the queue. This processing is called First In First Out or FIFO.

During the processing, the queue can be dynamically changed, i.e. processed elements are removed from the queue, and new elements are added to the queue.

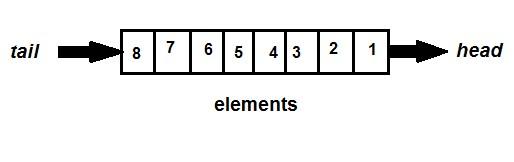
In the Java Collections Framework, Queue is the main interface, and there are four sub interfaces: Deque, BlockingDeque, BlockingQueue, and TransferQueue.

Except the Deque interface which is in the java.util package, all others are organized in the java.util.concurrent package, which is designed for multi-threading or concurrent programming.

Characteristics of Queue:

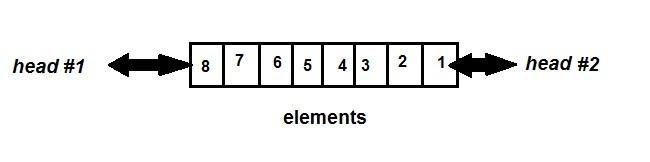


Basically, a queue has a head and a tail. New elements are added to the tail, and to-be-processed elements are picked from the head. The following picture illustrates a typical queue:



Elements in the queue are maintained by their insertion order. The Queue interface abstracts this kind of queue.

Another kind of queue is double ended queue, or deque. A deque has two heads, allowing elements to be added or removed from both ends. The following picture illustrates this kind of queue:



The Deque interface abstracts this kind of queue, and it is a sub interface of the Queue interface. And the LinkedList class is a well-known implementation. Some implementations accept null elements, some do not.

Queue does allow duplicate elements, because the primary characteristic of queue is maintaining elements by their insertion order. Duplicate elements in terms of equals contract are considered distinct in terms of queue, as there is no two elements having same ordering.

Additionally, the Java Collection Framework provides the BlockingQueue interface that abstracts queues which can be used in concurrent (multi-threading) context.

A blocking queue waits for the queue to become non-empty when retrieving an element, and waits for space become available in the queue when storing an element.

Similarly, the BlockingDeque interface is blocking queue for double ended queues.

Behaviors of Queue:



Due to Queue’s nature, the key operations that differentiate Queue from other collections are extraction and inspection at the head of the queue.

For deques, the extraction and inspection can be processed on both ends.

And because the Queue interface extends the Collection interface, all Queue implementations provide core operations of a collection like add(), contains(),remove(), clear(), isEmpty(), etc.

And keep in mind that, with queues, operations on the head are fastest (e.g. offer() and remove()), whereas operations on middle elements are slow (e.g.contains(obj) and remove(obj)).

Queue’s Interfaces:



Queue is the super interface of the queue branch in the Java Collection Framework. Under it, there are the following sub interfaces:

* **Deque**: abstracts a queue that has two heads. A deque allows adding or removing elements at both ends.
* **BlockingQueue**: abstracts a type of queues that waits for the queue to be non-empty when retrieving an element, and waits for space to become available in the queue when storing an element.
* **BlockingDeque**: is similar to BlockingQueue, but for double ended queues. It is sub interface of the BlockingQueue.

And since Java 7, the BlockingQueue interface has a new sub interface called TransferQueue, which is a specialized BlockingQueue, which waits for another thread to retrieve an element in the queue.

Major Queue’s Implementations:



The Java Collection framework provides many implementations, mostly for the BlockingQueue interface. Below I name few which are used commonly.

Queue implementations are grouped into two groups: general-purpose and concurrent implementations.

-General-purpose Queue implementations:

* + **LinkedList**: this class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends, plus accessing elements by index.

* + **PriorityQueue**: this queue orders elements according to their natural ordering, or by a Comparator provided at construction time. Consider using a PriorityQueue when you want to take advantages of natural ordering and fast adding elements to the tail and fast removing elements at the head of the queue.

* + **ArrayDeque**: a simple implementation of the Deque interface. Consider using an ArrayDeque when you want to utilize features of a double ended queue without list-based ones (simpler than a LinkedList).

Concurrent Queue implementations:

* + **ArrayBlockingQueue**: this is a blocking queue backed by an array. Consider using an ArrayBlockingQu0eue when you want to use a simple blocking queue that has limited capacity (bounded).

* + **PriorityBlockingQueue**: Use this class when you want to take advantages of both PriorityQueue and BlockingQueue.

* + **DelayQueue**: a time-based scheduling blocking queue. Elements added to this queue must implement the Delayed interface. That means an element can only be taken from the head of the queue when its delay has expired.

# Mastering Queue Collection in Java



## Queue API Structure

Before giving you code examples about using Queue collections, I think it’s necessary to understand deeper about the API Structure of Queue collection, as queues have very different characteristics than other types of collection.

Because the Queue interface extends the Collection interface, al Queue implementations have basic operations of a collection:

* Single operations: add(e),contains(e), iterator(), clear(), isEmpty(), size() and toArray().
* Bulk operations: addAll(), containsAll(), removeAll() and retainAll().

Understanding Queue interface’s API Structure:



Basically, Queue provides three primary types of operations which differentiate a queue from others:

1. **Insert**: adds an element to the tail of the queue.
2. **Remove**: removes the element at the head of the queue.
3. **Examine**: returns, but does not remove, the element at the head of the queue. And for each type of operation, there are two versions:

▪ The first version throws an exception if the operation fails, e.g. could not add element when the queue is full. ▪ The second version returns a special value (either null or false, depending on the operation).

The following table summarizes the main operations of the Queue interface:

| **Type of operation** | **Throws exception** | **Returns special value** |
| --- | --- | --- |
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| Examine | element() | peek() |

Understanding Deque interface’s API Structure:



As you know, the Deque interface abstracts a double ended queue with two ends (first and last), so its API is structured around this characteristic.

A Deque implementation provides the xxxFirst() methods that operate on the first element, and the xxxLast() methods that operate on the last element. The following table summarizes the API structure of Deque:

| **Type of operation** | **First element** | **Last element** |
| --- | --- | --- |
| Insert | addFirst(e) offerFirst(e) | addLast(e) offerLast(e) |
| Remove | removeFirst() pollFirst() | removeLast() pollLast() |
| Examine | getFirst() peekFirst() | getLast() peekLast() |

Understanding BlockingQueue interface’s API Structure:



A blocking queue is designed to wait for the queue to become non-empty when retrieving an element (the put(e) method), and wait for space to become available in the queue when storing an element (the take() method).

In addition, a blocking queue provides specialized operations that can wait up to a specified duration when inserting and removing an element.

The following table summarizes the API structure of BlockingQueue interface:

| **Type of operation** | **Throws exception** | **special value** | **blocks** | **times out** |
| --- | --- | --- | --- | --- |
| Insert | add(e) | offer(e) | put(e) | offer(e, time, unit) |
| Remove | remove() | poll() | take() | poll(time, unit) |
| Examine | element() | peek() | N/A | N/A |

Understanding BlockingDeque interface’s API Structure:



Similarly, a BlockingDeque is a specialized BlockingQueue for double ended queue with two ends (head and tail). Its API is in scheme of xxxFirst() methods operating on the first element and xxxLast() methods operating on the last element.

The following table summarizes the API structure of BlockingDeque:

| **First Element (head)** | |  |  |  |
| --- | --- | --- | --- | --- |
|  | Throws exception | special value | blocks | times out |
| **Insert** | addFirst(e) | offerFirst(e) | putFirst(e) | offerFirst(e, time, unit) |
| **Remove** | removeFirst() | pollFirst() | takeFirst() | pollFirst(time, unit) |
| **Examine** | getFirst() | peekFirst() | N/A | N/A |

| **Last Element (tail)** |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Throws exception | special value | blocks | times out |
| **Insert** | addLast(e) | offerLast(e) | putLast(e) | offerLast(e, time, unit) |
| **Remove** | removeLast() | pollLast() | takeLast() | pollLast(time, unit) |
| **Examine** | getLast() | peekLast() | N/A | N/A |

# Performing Operations on Queue Collection



Let’s go through various code examples to understand how to use Queue collections in daily coding. In the upcoming examples, I use different implementations like LinkedList, ArrayDeque, PriorityQueue,ArrayBlockingQueue, etc.

# 1. Creating a New Queue Instance



As a best practice, it’s recommended to use generic type and interface as reference type when creating a new collection. For queues, depending on the need of a particular type (queue, deque and blocking queue), use the corresponding interface as the reference type.

For example, the following statements create 3 different types of queues:

Queue<String> namesQueue = new LinkedList<>();

Deque<Integer> numbersDeque = new ArrayDeque<>();

BlockingQueue<String> waitingCustomers = new ArrayBlockingQueue<>(100);

Most Queue implementations do not have restriction on capacity (unbounded queues), except

the ArrayBlockingQueue, LinkedBlockingQueue and LinkedBlockingDeque classes. The following statement creates an ArrayBlockingQueue with fixed capacity of 200 elements:

BlockingQueue<String> waitingCustomers = new ArrayBlockingQueue<>(200);

Also remember that we can use the copy constructor to create a new Queue instance from another collection. For example:

List<String> listNames = Arrays.asList("Alice", "Bob", "Cole", "Dale", "Eric", "Frank"); Queue<String> queueNames = new LinkedList<>(listNames); System.out.println(queueNames); Output:

[Alice, Bob, Cole, Dale, Eric, Frank]

# 2. Adding New Elements to the Queue



To insert an element to the tail of the queue, we can use either the add() or offer() method. The following code adds two elements to a linked list:

Queue<String> queueNames = new LinkedList<>(); queueNames.add("Mary"); queueNames.add("John");

When using an unbounded queue (no capacity restriction), the add() and offer() methods do not show the difference. However, when using a bounded queue, the add() method will throw an exception if the queue is full, while the offer() method returns false. The following example illustrates this difference: Queue<Integer> queueNumbers = new ArrayBlockingQueue<>(3); queueNumbers.add(1); queueNumbers.add(2); queueNumbers.add(3);

queueNumbers.add(4);// this line throws exception

The last line throws java.lang.IllegalStateException: Queue full because we declare the queue with capacity of 3 elements. Hence adding the 4th element results in an exception.

However, we are safe when using the offer() method, as shown in the following code snippet:

Queue<Integer> queueNumbers = new ArrayBlockingQueue<>(3);

System.out.println(queueNumbers.offer(1));

System.out.println(queueNumbers.offer(2));

System.out.println(queueNumbers.offer(3));

System.out.println(queueNumbers.offer(4)); Output:

true true true false

The following code snippet adds an element to the head and an element to the tail of a double ended queue (notice the type of the interface is used):

Deque<String> queueNames = new ArrayDeque<>(); queueNames.offer("Katherine"); queueNames.offer("Bob"); queueNames.addFirst("Jim"); queueNames.addLast("Tom"); System.out.println(queueNames); Output:

[Jim, Katherine, Bob, Tom]

For blocking queue, use the put(e) or offer(e, time, unit) in case you want the current thread to wait until space becomes available in the queue. For example:

BlockingQueue<Integer> queueNumbers = new ArrayBlockingQueue<>(100); try {

queueNumbers.put(2000); } catch (InterruptedException ie) { ie.printStackTrace();

}

# 3. Removing the Head of the Queue



A Queue provides the remove() and poll() methods that allow us to pick the element at the head and remove it from the queue. And you should understand the difference between these two methods.

The remove() method returns the head element or throws an exception if the queue is empty. For example: Queue<String> queueCustomers = new LinkedList<>(); queueCustomers.offer("Jack");

String next = queueCustomers.remove();

System.out.println("Next customer is: "+ next); next = queueCustomers.remove(); // throws exception

Here, the queue has only one element, so the first call to remove() working fine. However the subsequent invocation results in java.util.NoSuchElementException because the queue becomes empty.

In contrast, the poll() method returns null if the queue is empty, as shown in the following example: Queue<String> queueCustomers = new LinkedList<>(); queueCustomers.offer("Jack");

System.out.println("next: " + queueCustomers.poll());

System.out.println("next: " + queueCustomers.poll()); // returns null Output:

next: Jack next: null

The following example removes the head element and tail element from a deque: Deque<String> queueCustomers = new ArrayDeque<>(); queueCustomers.offer("Bill"); queueCustomers.offer("Kim"); queueCustomers.offer("Lee"); queueCustomers.offer("Peter"); queueCustomers.offer("Sam");

System.out.println("Queue before: " + queueCustomers);

System.out.println("First comes: " + queueCustomers.pollFirst()); System.out.println("Last comes: " + queueCustomers.pollLast()); System.out.println("Queue after: " + queueCustomers); Output:

Queue before: [Bill, Kim, Lee, Peter, Sam]

First comes: Bill

Last comes: Sam

Queue after: [Kim, Lee, Peter]

For a blocking queue, use the take()or poll(time, unit) methods in case you want the current thread to wait until an element becomes available. For example:

BlockingQueue<String> queueCustomers = new ArrayBlockingQueue<>(100); queueCustomers.offer("Kathe"); try {

String nextCustomer = queueCustomers.take();

} catch (InterruptedException ie) { ie.printStackTrace();

}

# 4. Examining the Head of the Queue



In contrast to the remove() method, the examine methods element() and peek() return (but do not remove) the head of the queue. So consider using these methods in case you just want to check what is currently in the head element without modifying the queue.

Also, you need to know the difference between the element() and peek() methods:

The element() method throws an exception in case the queue is empty, whereas the peek() method returns null. For example:

Queue<String> queueCustomers = new PriorityQueue<>(); queueCustomers.offer("Jack");

System.out.println("who's next: " + queueCustomers.poll()); // this returns null in case the queue is empty

System.out.println("who's next: " + queueCustomers.peek());

// this throws exception in case the queue is empty

System.out.println("who's next: " + queueCustomers.element());

For a deque, use the getFirst() or peekFirst() methods to examine the first element, and getLast() or peekLast() to examine the last element. Here’s an example: Deque<Integer> queueNumbers = new ArrayDeque<>(); queueNumbers.add(10); queueNumbers.add(20); queueNumbers.add(30); queueNumbers.add(40);

System.out.println("first: " + queueNumbers.getFirst());

System.out.println("last: " + queueNumbers.peekLast()); **There’s no method for examining a blocking queue.**

# 5. Iterating over Elements in the Queue



We can use the enhanced for loop, iterator and forEach() method to iterate over elements in the queue. The following code snippet illustrates how to iterate a linked list using the enhanced for loop: Queue<String> queueNames = new LinkedList<>(); queueNames.add("Dale"); queueNames.add("Bob"); queueNames.add("Frank"); queueNames.add("Alice"); queueNames.add("Eric"); queueNames.add("Cole"); queueNames.add("John"); for (String name : queueNames) {

System.out.println(name);

} Output:

Dale

Bob

Frank

Alice

Eric

Cole

John

More simply, using Lambda expression with forEach() method in Java 8:

queueNames.forEach(name -> System.out.println(name));

The following example iterates over elements of a PriorityQueue which sorts elements by natural ordering: Queue<String> queueNames = new PriorityQueue<>(); queueNames.add("Dale"); queueNames.add("Bob"); queueNames.add("Frank"); queueNames.add("Alice"); queueNames.add("Eric"); queueNames.add("Cole"); queueNames.add("John");

queueNames.forEach(name -> System.out.println(name)); Output:

Alice

Bob

Cole

Dale

Eric

Frank John

As you can see in the output, the elements are sorted in the alphabetic order (natural ordering of Strings).

**NOTE:** Pay attention when using an iterator of a PriorityQueue, because it is not guaranteed to traverse the elements of the priority queue in any particular order.

# 6. Concurrent Queues



All implementations of BlockingQueue are thread-safe. The following implementations are not:

* LinkedList
* ArrayDeque
* PriorityQueue

When you want to use a synchronized linked list, use the following code:

List list = Collections.synchronizedList(new LinkedList<>());

And consider using the PriorityBlockingQueue instead of the PriorityQueue when you want to use a synchronized priority queue.

# Producer - Consumer Example Using Queue



Typically, queue is used to implement producer-consumer scenarios. The following kinds of program will need to use queue:

* **Chat applications:** Messages are put into a queue. When you are sending a message, you are the producer; and your friend who reads the message, is the consumer. Messages need to be kept in queue because of network latency. Imagine network connection dropped when you are trying to send a message. In this case, the message is still in the queue, awaiting the receiver to consume upon the connection becomes available.
* **Online help desk applications:** Imagine a company has 5 persons working as customer support staffs. They chat with clients through a help desk application. They can talk with maximum 5 clients at a time, so other clients will be queued up. When a staff finishes serving a client, the next client in the queue is served next.
* **Real-time processing applications such as screen recorder**. The logic behind this kind of application is there are two threads working concurrently: The producer thread captures screenshots constantly and puts the images into a queue; the consumer thread takes the images from the queue to process the video.
* **And much more.**

Above I name only few types of application in which we need to use queue. Remember using queue when you need to implement producer-consumer processing.

In Java, using a BlockingQueue implementation is a good choice, as its put(e) method let the producer thread waits for space to become available in the queue, and its take() method let the consumer thread waits for an element become available in the queue.

The following is a pseudo-code of a typical producer-consumer application:

class Producer implements Runnable { private final BlockingQueue queue; Producer(BlockingQueue q) { queue = q; } public void run() {

while (true) { queue.put(produce()); }

}

Object produce() { ... }

}

class Consumer implements Runnable { private final BlockingQueue queue; Consumer(BlockingQueue q) { queue = q; } public void run() {

while (true) { consume(queue.take()); }

}

void consume(Object x) { ... }

}

class Program { void main() {

BlockingQueue q = new SomeBlockingQueueImplementation();

Producer p = new Producer(q);

Consumer c1 = new Consumer(q);

Consumer c2 = new Consumer(q);

new Thread(p).start(); new Thread(c1).start(); new Thread(c2).start();

}

}

Here, the Producer class is a thread which constantly produces objects and put them into the queue. In practice, we should specify condition to exit the loop, such as closing/shutdown the program or a maximum number of objects reached.

The Consumer class is another thread which constantly takes objects from the queue to process. In practice, we should specify condition to stop this thread by checking the queue for a special object (null, false or special value), for example:

while (true) {

Integer number = queue.take(); if (number == -1) { break; }

consume(number);

}

And in this case, the producer is responsible to put this special object into the queue to indicate there’s no more elements to process.

For you reference, here’s a working example that gives you the idea. It’s a program that creates one producer thread and one consumer thread:

Producer:

import java.util.\*; import java.util.concurrent.\*;

public class Producer implements Runnable {

private BlockingQueue<Integer> queue; public Producer (BlockingQueue<Integer> queue) { this.queue = queue;

}

public void run() { try {

for (int i = 0; i < 10; i++) { queue.put(produce());

Thread.sleep(500);

}

queue.put(-1); // indicates end of producing System.out.println("Producer STOPPED.");

} catch (InterruptedException ie) { ie.printStackTrace();

}

}

private Integer produce() {

Integer number = new Integer((int) (Math.random() \* 100));

System.out.println("Producing number => " + number); return number;

}

}

Consumer:

import java.util.\*; import java.util.concurrent.\*; public class Consumer implements Runnable { private BlockingQueue<Integer> queue; public Consumer(BlockingQueue<Integer> queue) { this.queue = queue;

}

public void run() { try {

while (true) {

Integer number = queue.take(); if (number == -1) { break;

}

consume(number);

Thread.sleep(1000);

}

System.out.println("Consumer STOPPED.");

} catch (InterruptedException ie) { ie.printStackTrace();

} }

private void consume(Integer number) {

System.out.println("Consuming number <= " + number);

}

}

Test Program:

import java.util.\*; import java.util.concurrent.\*; public class ProducerConsumerTest {

public static void main(String[] args) {

BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(20);

Thread producer = new Thread(new Producer(queue));

Thread consumer = new Thread(new Consumer(queue)); producer.start(); consumer.start();

}

}

Compile and run this program would print the following output:

Producing number => 21

Consuming number <= 21

Producing number => 90

Producing number => 51

Consuming number <= 90

Producing number => 23

Producing number => 61

Consuming number <= 51

Producing number => 63

Producing number => 75

Consuming number <= 23

Producing number => 99

Consuming number <= 61

Producing number => 59

Producing number => 31 Producer STOPPED.

Consuming number <= 63

Consuming number <= 75

Consuming number <= 99

Consuming number <= 59

Consuming number <= 31 Consumer STOPPED.

# Understanding SortedSet and TreeSet



You know, TreeSet does not only implement the Set interface, it also implements the SortedSet and NavigableSet interfaces. Therefore, besides inheriting behaviors of a typical Set, TreeSet also inherits behaviors of SortedSet and NavigableSet. The following picture illustrates the API hierarchy:

Understanding SortedSet:



The key characteristic of a SortedSet is that, it sorts elements according to their natural ordering or by a specified comparator. So considering using a TreeSet when you want a collection that satisfies the following conditions:

* Duplicate elements are not allowed.
* Elements are sorted by their natural ordering (default) or by a specified comparator.

Here’s an example illustrates this characteristic of a SortedSet: SortedSet<Integer> setNumbers = new TreeSet<>();

setNumbers.addAll(Arrays.asList(2, 1, 4, 3, 6, 5, 8, 7, 0, 9)); System.out.println("Sorted Set: " + setNumbers); Output:

Sorted Set: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Here, we add elements of an array list to a TreeSet, and as you can see, the duplicate elements are removed and they are sorted by alphanumeric order (natural ordering of numbers).

In addition to basic collection operations and normal set operations, the SortedSet provides the following types of operations:

* **Range view**: extracts a portion of the set, i.e. a range.
* **Endpoints**: returns the first and the last element in the sorted set.
* **Comparator** **access**: returns the comparator, if an, used to sort the set.

Hence the following interface abstracts a SortedSet:

public interface SortedSet<E> extends Set<E> {

// Range-view

SortedSet **subSet**(E fromElement, E toElement);

SortedSet **headSet**(E toElement);

SortedSet **tailSet**(E fromElement);

// Endpoints

E **first**();

E **last**();

// Comparator access

Comparator<? super E> **comparator**(); }

Let’s look at each type of operation in details.

Range view operations:

* + SortedSet **subSet(E fromElement, E toElement)**: returns a sorted set which is a portion of the set whose elements range from fromElement, inclusive, to toElement, exclusive.

* + SortedSet **headSet(E toElement)**: returns a sorted set which is a portion of the set whose elements are

strictly less than toElement.

* + SortedSet **tailSet(E fromElement)**: returns a sorted set which is a portion of the set whose elements are greater than or equal to fromElement.

Endpoint operations:

* + E **first():** returns the first (lowest) element currently in the set.
* + E **last()**: returns the last (highest) element currently in the set.

Comparator access:

+ **comparator()**: returns the comparator used to order the elements in the set, or null if this set uses the natural ordering of its elements.

Code Examples:

The following code example demonstrates how these operations work with a **TreeSet** implementation: SortedSet<Integer> setNumbers = new TreeSet<>();

setNumbers.addAll(Arrays.asList(2, 1, 4, 3, 6, 5, 8, 7, 0, 9));

System.out.println("Original Set: " + setNumbers);

Integer first = setNumbers.first();

System.out.println("First element: " + first);

Integer last = setNumbers.last();

System.out.println("Last element: " + last);

SortedSet<Integer> subSet = setNumbers.subSet(3, 7);

System.out.println("Sub Set: " + subSet);

SortedSet<Integer> headSet = setNumbers.headSet(5); System.out.println("Head Set: " + headSet);

SortedSet<Integer> tailSet = setNumbers.tailSet(5);

System.out.println("Tail Set: " + tailSet);

Comparator comparator = setNumbers.comparator();

System.out.println("Sorted by natural ordering? " + (comparator == null)); Output:

Original Set: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

First element: 0

Last element: 9

Sub Set: [3, 4, 5, 6]

Head Set: [0, 1, 2, 3, 4]

Tail Set: [5, 6, 7, 8, 9]

Sorted by natural ordering? true

The following code snippet shows how to use a comparator:

class ZtoAComparator implements Comparator<String> { public int compare(String s1, String s2) { return s2.compareTo(s1);

}

}

SortedSet<String> setStrings = new TreeSet<>(new ZtoAComparator()); setStrings.add("Banana"); setStrings.add("Apple"); setStrings.add("Grape"); setStrings.add("Lemon"); setStrings.add("Watermelon");

System.out.println(setStrings); Output:

[Watermelon, Lemon, Grape, Banana, Apple]

As you see, the specified comparator sorts the elements into descending order.

If you use Java 8, use Lambda expression to simplify the comparator class like this:

SortedSet<String> setStrings = new TreeSet<>((s1, s2) -> s2.compareTo(s1));

# Understanding NavigableSet and TreeSet



Understand the NavigabeSet interface in the Java Collections Framework with code examples using TreeSet. Besides Set and SortedSet, TreeSet also implements NavigableSet.

Understanding NavigableSet:

NavigableSet is a sub interface of the SortedSet interface, so it inherits all SortedSet’s behaviors like range view, endpoints and comparator access. In addition, the NavigableSet interface provides navigation methods and descending iterator that allows the elements in the set can be traversed in descending order.

Let’s look at each new method defined by this interface in details.

* lower(E): returns the greatest element which is less than the specified element E, or null if there is no such element.
* floor(E): returns the greatest element which is less than or equal to the given element E.
* ceiling(E): returns the least element which is greater than or equal to the given element E.
* higher(E): returns the least element which is strictly greater than the specified element E.
* descendingSet(): returns a reverse order view of the elements contained in the set.
* descendingIterator(): returns an iterator that allows traversing over elements in the set in descending order.
* pollFirst(): retrieves and removes the first (lowest) element, or returns null if the set is empty. ▪ pollLast(): retrieves and removes the last (highest) element, or returns null if the set is empty.

Furthermore, the NavigableSet interface overloads the methods headSet(), subSet() and tailSet() from the SortedSet interface, which accepts additional arguments describing whether lower or upper bounds are inclusive versus exclusive:

* headSet(E toElement, boolean inclusive)
* subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) ▪ tailSet(E fromElement, boolean inclusive) Now, let’s look at some code examples.

Code Examples:

The following example shows how to obtain a reverse order set from the original one: NavigableSet<Integer> setNumbers1 = new TreeSet<>();

setNumbers1.addAll(Arrays.asList(4, 8, 3, 9, 1, 6, 4, 5, 3, 2, 7, 8, 0));

NavigableSet<Integer> setNumbers2 = setNumbers1.descendingSet();

System.out.println("Set Numbers 1: " + setNumbers1);

System.out.println("Set Numbers 2: " + setNumbers2); Output:

Set Numbers 1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Set Numbers 2: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

The following example illustrates how to obtain the descending iterator from a navigable set: NavigableSet<String> setFruits = new TreeSet<>(); setFruits.add("Banana"); setFruits.add("Apple"); setFruits.add("Orange"); setFruits.add("Grape"); setFruits.add("Mango");

System.out.println("Set Fruits: " + setFruits);

Iterator<String> descIterator = setFruits.descendingIterator(); System.out.print("Fruits by descending order: "); while (descIterator.hasNext()) { System.out.print(descIterator.next() + ", "); } Output:

Set Fruits: [Apple, Banana, Grape, Mango, Orange]

Fruits by descending order: Orange, Mango, Grape, Banana, Apple,

The following example demonstrates the benefits of using navigation methods like lower(), higher(), ceiling() and floor(); and range view methods like headSet(), subSet() and tailSet().

Given the following entity class:

class Employee {

String name; int salary;

public Employee(int salary) { this.salary = salary;

}

public Employee(String name, int salary) {  [this.name](http://this.name/) = name; this.salary = salary;

}

public String toString() {

return [this.name](http://this.name/) + "-" + salary;

}

public String getName() { return [this.name;](http://this.name/)

}

public Integer getSalary() { return new Integer(this.salary);

}

public boolean equals(Object obj) { if (obj instanceof Employee) { Employee another = (Employee) obj; if (this.name.equals([another.name)](http://another.name/) && this.salary == another.salary) { return true;

} } return false;

}

public int hashCode() {

return 31 \* name.hashCode() + salary;

}

}

Note that this class overrides the equals() and hashCode() methods based on employee’s name and salary. The following comparator class compares two employees based on their salary:

public class EmployeeComparator implements Comparator<Employee> { public int compare(Employee emp1, Employee emp2) { return emp1.getSalary().compareTo(emp2.getSalary()); }

}

We add 8 employees into a navigable set like this:

NavigableSet<Employee> setEmployees = new TreeSet<>(new EmployeeComparator()); setEmployees.add(new Employee("Tom", 80000)); setEmployees.add(new Employee("Jack", 35000)); setEmployees.add(new Employee("Jim", 62500)); setEmployees.add(new Employee("Peter", 58200)); setEmployees.add(new Employee("Mary", 77000)); setEmployees.add(new Employee("Jane", 69500)); setEmployees.add(new Employee("David", 54000)); setEmployees.add(new Employee("Sam", 82000));

System.out.println("Employees: " + setEmployees); Output:

Employees: [Jack-35000, David-54000, Peter-58200, Jim-62500, Jane-69500, Mary-77000, Tom80000, Sam-82000]

Here, an employee object is printed with name and salary.

* **Using the higher() method, we can know the employee whose salary is higher than the employee ‘Tom’:**

Employee Tom = new Employee("Tom", 80000);

Employee emp1 = setEmployees.higher(Tom); if (emp1 != null) { System.out.println("The employee whose salary > Tom: " + emp1); } Output:

The employee whose salary > Tom: Sam-82000

**NOTE**: to allow this kind of search possible, the entity class must correctly override the equals() and hashCode() method, as shown in the Employee class above.

* **Using the lower() method, we can know the employee whose salary is less than the employee Tom:** Employee emp2 = setEmployees.lower(Tom); if (emp2 != null) {

System.out.println("The employee whose salary < Tom: " + emp2); } Output:

The employee whose salary < Tom: Mary-77000

* **Using the ceiling() method, we can know the employee whose salary is greater than 60,000 USD/year like this:** Employee emp3 = setEmployees.ceiling(new Employee(60000)); if (emp3 != null) {

System.out.println("The employee whose >= 60000: " + emp3); } Output:

The employee whose >= 60000: Jim-62500

* **Using the floor() method, we can know the employee whose salary is less than 40,000 USD like this:** Employee emp4 = setEmployees.floor(new Employee(40000)); if (emp4 != null) {

System.out.println("The employee whose <= 40000: " + emp4); } Output:

The employee whose <= 40000: Jack-35000

* **With the tailSet() method, we can know the employees who are high paid (salary > 70,000 USD):**

SortedSet<Employee> highPaidEmployees = setEmployees.tailSet(new Employee(70000)); System.out.println("High paid employees: " + highPaidEmployees); Output:

High paid employees: [Mary-77000, Tom-80000, Sam-82000]

* **With the headSet() method, we can know the employees who are low paid (under 40,000USD/year):**

SortedSet<Employee> lowPaidEmployees = setEmployees.headSet(new Employee(60000)); System.out.println("Low paid employees: " + lowPaidEmployees); Output:

Low paid employees: [Jack-35000, David-54000, Peter-58200]

* **With the subSet() method, we can know the employees who are good paid (salary is between 60,000 and 70,000):**

SortedSet<Employee> goodPaidEmployees = setEmployees.subSet(new Employee(60000), new

Employee(70000));

System.out.println("Good paid employees: " + goodPaidEmployees);

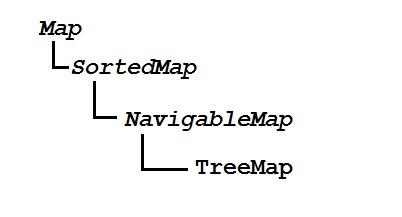
Output:

Good paid employees: [Jim-62500, Jane-69500]

# Understanding SortedMap and TreeMap



TreeMap doesn’t only implement the Map interface, it also implements the SortedMap and NavigableMap interfaces. Therefore, besides the behaviors inherited from the Map, TreeMap also inherits the behaviors defined by SortedMap and NavigableMap. The following picture depicts the API hierarchy of TreeMap:



Understanding SortedMap:



The main characteristic of a SortedMap is that, it orders the keys by their natural ordering, or by a specified comparator. So consider using a TreeMap when you want a map that satisfies the following criteria:

* null key or null value are not permitted.
* The keys are sorted either by natural ordering or by a specified comparator.

The following example realizes the concept of a SortedMap: SortedMap mapDomains = new TreeMap<>(); mapDomains.put(".com", "International"); mapDomains.put(".us", "United States"); mapDomains.put(".uk", "United Kingdom"); mapDomains.put(".jp", "Japan"); mapDomains.put(".au", "Australia");

System.out.println(mapDomains); Output:

{.au=Australia, .com=International, .jp=Japan, .uk=United Kingdom, .us=United States}

Here, this map contains mappings of domain=country, and as we see in the output, the domains (keys) are sorted by alphabetic order (natural ordering of Strings).

Besides the operations inherited from the Map interface, the SortedMap also defines the following operations:

* **Range view**: returns a sub sorted map whose keys fall within a range of keys in the original map.
* **Endpoints**: returns the first or last key in the sorted map.
* **Comparator access**: returns the comparator (implements the Comparator interface), if any, used to sort the map.

Hence the following code is the definition of a SortedMap:

public interface SortedMap extends Map{

Comparator comparator();

SortedMap subMap(K fromKey, K toKey);

SortedMap headMap(K toKey);

SortedMap tailMap(K fromKey);

K firstKey();

K lastKey();

}

Let’s look at each type of operation in details.

Range View Operations:

* **+ subMap(K fromKey, K toKey):** returns a sorted map whose keys range from fromKey, inclusive, to toKey, exclusive.
* **+ headMap(K toKey):** returns a sorted map whose keys are strictly less than toKey.
* **+ tailMap(K fromKey):** returns a sorted map whose keys are greater than or equal to fromKey.

Endpoint operations:

* **+ firstKey():** returns the first (lowest) key currently in the map.
* **+ lastKey():** returns the last (highest) key currently in the map.

Comparator access:

* **+ comparator():** returns the comparator used to order the keys in the map, or returns null if this map uses the natural ordering of its keys.

The following code example demonstrates how to work with these operations on a TreeMap: SortedMap mapHttpStatus = new TreeMap<>(); mapHttpStatus.put(100, "Continue"); mapHttpStatus.put(200, "OK");

mapHttpStatus.put(300, "Multiple Choices"); mapHttpStatus.put(400, "Bad Request"); mapHttpStatus.put(401, "Unauthorized"); mapHttpStatus.put(402, "Payment Required"); mapHttpStatus.put(403, "Forbidden"); mapHttpStatus.put(404, "Not Found"); mapHttpStatus.put(500, "Internal Server Error"); mapHttpStatus.put(501, "Not Implemented"); mapHttpStatus.put(502, "Bad Gateway");

System.out.println("All key-value pairs: "); for (Integer code : mapHttpStatus.keySet()) {

System.out.println(code + " -> " + mapHttpStatus.get(code));

}

System.out.println();

Integer firstKey = mapHttpStatus.firstKey();

String firstValue = mapHttpStatus.get(firstKey);

System.out.println("First status: " + firstKey + " -> " + firstValue); System.out.println();

Integer lastKey = mapHttpStatus.lastKey();

String lastValue = mapHttpStatus.get(lastKey);

System.out.println("Last status: " + lastKey + " -> " + lastValue);

System.out.println();

SortedMap map4xxStatus = mapHttpStatus.subMap(400, 500); System.out.println("4xx Statuses: "); for (Integer code : map4xxStatus.keySet()) {

System.out.println(code + " -> " + map4xxStatus.get(code));

}

System.out.println();

SortedMap mapUnder300Status = mapHttpStatus.headMap(300); System.out.println("Statuses < 300: "); for (Integer code : mapUnder300Status.keySet()) {

System.out.println(code + " -> " + mapUnder300Status.get(code));

}

System.out.println();

SortedMap mapAbove500Status = mapHttpStatus.tailMap(500); System.out.println("Statuses > 500: "); for (Integer code : mapAbove500Status.keySet()) {

System.out.println(code + " -> " + mapAbove500Status.get(code));

}

Comparator comparator = mapHttpStatus.comparator();

System.out.println("Sorted by natural ordering? " + (comparator == null)); Output:

All key-value pairs:

100 -> Continue

200 -> OK

300 -> Multiple Choices

1. -> Bad Request
2. -> Unauthorized
3. -> Payment Required
4. -> Forbidden
5. -> Not Found
6. -> Internal Server Error
7. -> Not Implemented
8. -> Bad Gateway

First status: 100 -> Continue

Last status: 502 -> Bad Gateway 4xx Statuses:

1. -> Bad Request
2. -> Unauthorized
3. -> Payment Required
4. -> Forbidden 404 -> Not Found

Statuses < 300:

100 -> Continue

200 -> OK

Statuses > 500:

1. -> Internal Server Error
2. -> Not Implemented
3. -> Bad Gateway

Sorted by natural ordering? true

And the following example shows how to use a comparator:

SortedMap mapHttpStatus = new TreeMap<>(new ReverseComparator()); mapHttpStatus.put(100, "Continue"); mapHttpStatus.put(200, "OK"); mapHttpStatus.put(300, "Multiple Choices"); mapHttpStatus.put(400, "Bad Request"); mapHttpStatus.put(401, "Unauthorized"); mapHttpStatus.put(402, "Payment Required"); mapHttpStatus.put(403, "Forbidden"); mapHttpStatus.put(404, "Not Found"); mapHttpStatus.put(500, "Internal Server Error"); mapHttpStatus.put(501, "Not Implemented"); mapHttpStatus.put(502, "Bad Gateway");

for (Integer code : mapHttpStatus.keySet()) {

System.out.println(code + " -> " + mapHttpStatus.get(code)); }

Here’s the code of the comparator class:

class ReverseComparator implements Comparator { public int compare(Integer num1, Integer num2) { return num2.compareTo(num1);

}

} Output:

502 -> Bad Gateway

501 -> Not Implemented

500 -> Internal Server Error

404 -> Not Found

403 -> Forbidden

402 -> Payment Required

401 -> Unauthorized

400 -> Bad Request

300 -> Multiple Choices

200 -> OK

100 -> Continue

As you can see, this comparator sorts the map by the descending order of its keys.

In case you are working on Java 8, use Lambda expressions to shorten the comparator code like this:

SortedMap mapHttpStatus = new TreeMap<>((i1, i2) -> i2.compareTo(i1));

# Understanding NavigableMap and TreeMap



Understanding NavigableMap:

NavigableMap is sub interface of SortedMap interface, so it inherits all behaviors of a sorted map like range view, endpoints and comparator access operations. In addition, the NavigableMap interface provides navigation methods and descending views like NavigableSet. And due to the nature of a map which stores key-value mappings, the additional APIs are designed for both keys and key-value entries in the map.

Let’s look at these methods in details.

Operations on key-value mappings (entries):

* **lowerEntry(K key):** returns a key-value mapping associated with the greatest key strictly less than the given key.
* **floorEntry(K key):** returns a key-value mapping entry which is associated with the greatest key less than or equal to the given key.
* **ceilingEntry(K key):** returns an entry associated with the lest key greater than or equal to the given key.
* **higherEntry(K key):** returns an entry associated with the least key strictly greater than the given key.
* Note that all these methods return null if there is no such key.
* **firstEntry():** returns a key-value mapping associated with the least key in the map, or null if the map is empty.
* **lastEntry():** returns a key-value mapping associated with the greatest key in the map, or null if the map is empty.
* **descendingMap():** returns a reverse order view of the mappings contained in the map.
* **pollFirstEntry():** removes and returns a key-value mapping associated with the least key in the map, or null if the map is empty.
* **pollLastEntry():** removes and returns a key-value mapping associated with the greatest key in the map, or

null if the map is empty.

Operations on keys only:

* **lowerKey(K key):** returns the greatest key strictly less than the given key.
* **floorKey(K key):** returns the greatest key less than or equal to the given key.
* **ceilingKey(K key):** returns the least key greater than or equal to the given key.
* **higherKey(K key):** returns the least key strictly greater than the given key.
* **descendingKeySet():** returns a NavigableSet containing the keys in reverse order.

Note that all these methods return null if there is no such key.

Furthermore, the NavigableMap interface overloads the headMap(), subMap() and tailMap() methods of the SortedMap interface, which accept additional arguments describing whether lower or upper bounds are inclusive versus exclusive:

* + headMap(K toKey, boolean inclusive)
  + subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) - tailMap(K fromKey, boolean inclusive) Now, let’s look at some code examples.

NavigableMap Examples with TreeMap:



The following example shows us how to obtain the reverse order view of the keys in a map:

NavigableMap mapHttpStatus = new TreeMap<>();

mapHttpStatus.put(100, "Continue"); mapHttpStatus.put(200, "OK"); mapHttpStatus.put(400, "Bad Request"); mapHttpStatus.put(401, "Unauthorized"); mapHttpStatus.put(500, "Internal Server Error"); mapHttpStatus.put(501, "Not Implemented");

Set ascendingKeys = mapHttpStatus.keySet();

System.out.println("Ascending Keys: " + ascendingKeys);

Set descendingKeys = mapHttpStatus.descendingKeySet();

System.out.println("Descending Keys: " + descendingKeys); Output:

Ascending Keys: [100, 200, 400, 401, 500, 501]

Descending Keys: [501, 500, 401, 400, 200, 100]

Given the above map, the following code snippet gets a reverse order view of the map: NavigableMap descendingMap = mapHttpStatus.descendingMap(); for (Integer key : descendingMap.keySet()) {

System.out.println(key + " => " + descendingMap.get(key)); } Output:

501 => Not Implemented

500 => Internal Server Error

401 => Unauthorized

400 => Bad Request

200 => OK

100 => Continue

Using operations on keys is explained in the following example:

Integer lowerKey = mapHttpStatus.lowerKey(401); System.out.println("Lower key: " + lowerKey);

Integer floorKey = mapHttpStatus.floorKey(401);

System.out.println("Floor key: " + floorKey);

Integer higherKey = mapHttpStatus.higherKey(500);

System.out.println("Higher key: " + higherKey);

Integer ceilingKey = mapHttpStatus.ceilingKey(500);

System.out.println("Ceiling key: " + ceilingKey); Output:

Lower key: 400

Floor key: 401

Higher key: 501

Ceiling key: 500

And the following example demonstrates how to work with operations on key-value mapping entries:

Map.Entry firstEntry = mapHttpStatus.firstEntry();

System.out.println("First entry: " + firstEntry.getKey() + " => " + firstEntry.getValue());

Map.Entry lastEntry = mapHttpStatus.lastEntry();

System.out.println("Last entry: " + lastEntry.getKey() + " => " + lastEntry.getValue());

Map.Entry lowerEntry = mapHttpStatus.lowerEntry(401);

System.out.println("Lower entry: " + lowerEntry.getKey() + " => " + lowerEntry.getValue()); Map.Entry floorEntry = mapHttpStatus.floorEntry(401);

System.out.println("Floor entry: " + floorEntry.getKey() + " => " + floorEntry.getValue());

Map.Entry higherEntry = mapHttpStatus.higherEntry(500);

System.out.println("Higher entry: " + higherEntry.getKey() + " => " + higherEntry.getValue());

Map.Entry ceilingEntry = mapHttpStatus.ceilingEntry(500);

System.out.println("Ceiling entry: " + ceilingEntry.getKey() + " => " + ceilingEntry.getValue());

mapHttpStatus.pollFirstEntry(); mapHttpStatus.pollLastEntry();

System.out.println("\nMap after first and last entries were polled:"); for (Integer key : mapHttpStatus.keySet()) {

System.out.println(key + " => " + mapHttpStatus.get(key)); }

Output:

First entry: 100 => Continue

Last entry: 501 => Not Implemented

Lower entry: 400 => Bad Request

Floor entry: 401 => Unauthorized

Higher entry: 501 => Not Implemented Ceiling entry: 500 => Internal Server Error

Map after first and last entries were polled:

200 => OK

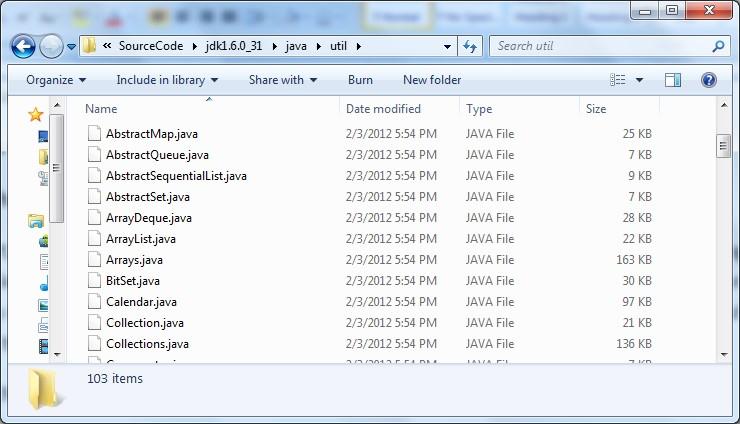
1. => Bad Request
2. => Unauthorized 500 => Internal Server Error

What are packages?



Packages are directory-based structures that group some related source files together. For example,

the java.util package in JDK groups all interfaces and classes in the Collections Framework such as Collection, List, ArrayList, Map, Set, HashMap, etc. The following screenshot shows a part of this package in Windows Explorer program:



To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.

Why are Packages?



* **Using packages allows us to avoid naming collisions:**

Imagine a situation in which two programmers write two classes that have same name, let’s say - Dog. If these two classes are used in a program, how to identify which Dog is which? So packages come to rescue: the 1st programmer puts his Dog class under a package called john.animal; and the 2nd programmer puts his Dog under tom.pets package.

When accessing these classes, we use their fully qualified names: john.animal.Dog and tom.pets.Dog.

In JDK, you can find some classes that have same name but in different packages, e.g. java.util.Date and java.sql.Date.

* **Packages facilitate the encapsulation feature in Java:**

Think packages like directories that isolate some classes from classes outside. In Java, we can use access modifiers to restrict access to some classes in a certain package. For example, the default access modifier (when no explicit access modifier is used) makes a class accessible only by others in the same package. Whereas the public access modifier makes a class visible and accessible by all classes regardless of packages.

* **Packages allow us to group some related classes together for better organization and management:**

For example, the java.util package contains only interfaces and classes which belong to the Collectionsframework; The javax.swing package contains only interfaces and classes which are related to Graphical UserInterface (GUI) components.

In practice, we tend to organize a complex application into packages for better organization and management, for example:

* + com.mycompany.model: contains entity classes.
  + com.mycompany.business: contain business classes.
  + com.mycompany.gui: contains GUI classes. etc.

# Packages



* A package does not mean only predefined classes; a package may contain user defined classes also.
* Packages can be compressed into JAR files for fast traversal in a network or to download from Internet
* With a single import statement, all the classes and interfaces can be obtained into our program
* Avoids namespace problems. Two classes of the same name cannot be put in the same package but can be placed in two different packages
* Access between the classes can be controlled. Using packages, restrictions can be imposed on the access of other package classes. Access specifiers work on package boundaries (between the classes of other packages) ▪ Packages and sub-packages are the easiest way to organize the classes

# Importing All/Single Class



* Packages have an advantage over header files of C-lang. A package allows importing a single class also instead of importing all. C-lang does not have this ease of getting one function from a header file import java.net.\*; // imports all the classes and interfaces import java.awt.event.\*; // imports all the classes and interfaces import java.net.Socket; // imports only Socket class import java.awt.event.WindowEvent; // imports only WindowEvent class
* Note: While importing a single class, asterisk (\*) should not be used

# Fully Qualified Class Name



* By placing the same class in two different packages, which Java permits, namespace problems can be solved. Namespace is the area of execution of a program in RAM ▪ The Date class exists in two packages –java.util and java.sql.
* Importing these two packages in a program gives ambiguity problem to the compiler. In the following program compiler gets ambiguity problem and is solved with fully-qualified name
* Note: When fully qualified name is included, importing the package is not necessary

| **Fully Qualified names** |
| --- |
| class PackageDemo1  {  public static void main(String args[])  { |

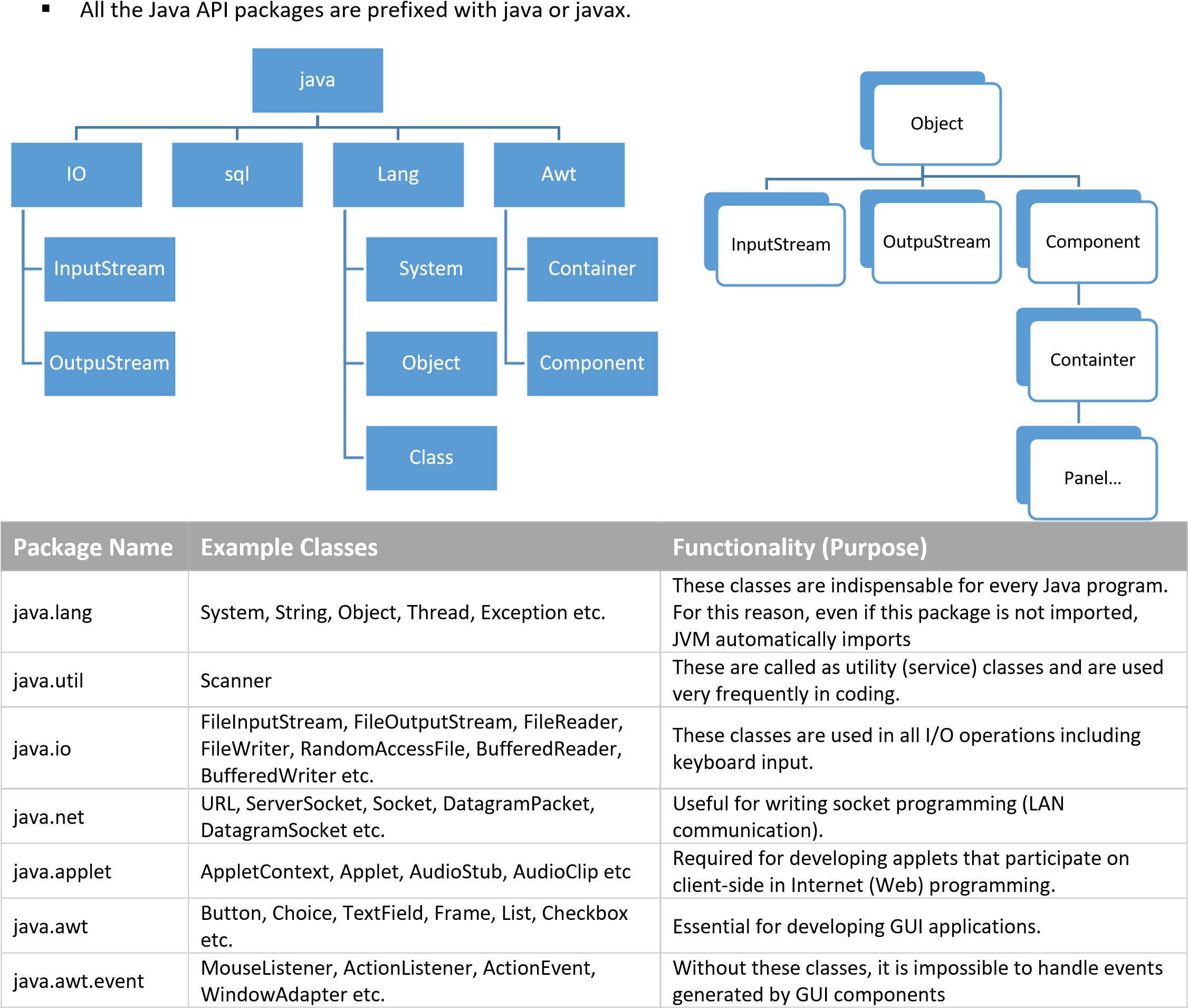
Program N0.

|  |  | System.out.println("\n\t Using util Package...."); java.util.Date d1 = new java.util.Date();    System.out.println("\n\t Using sql Package....");  java.sql.Date d2 = new java.sql.Date(1); |
| --- | --- | --- |
| } | } |  |

# Java Class Libraries – Java API



▪ All the classes and interfaces that come with the installation of JDK are put together are known as Java API (Application Programming Interface)



| **Date Class** |
| --- |

**Program N0.**

import java.util.Date;

public class PackageDemo\_154

{

public static void main(String args[])

{

Date today = new Date();

System.out.println("Today Particulars: " + today);

System.out.println("Hours Part of today: " + today.getHours());

System.out.println("Minutes part of today: " + today.getMinutes());

System.out.println("Seconds part of today: " + today.getSeconds());

System.out.println("Month part of today: " + today.getMonth());

System.out.println("Date part of today: " + today.getDate());

System.out.println("Day part of today: " + today.getDay());

System.out.println("Year part of today: " + today.getYear()); System.out.println("Milliseconds representation of today (from epoch time 01-01-1900: " + today.getTime());

}

}

# Math Class



* The java.lang.Math class includes many methods with which simple arithmetic operations can be done like finding the square root, rounding, trigonometric and logarithm functions.
* As these methods are defined as static, they can be used directly by the programmer without creating an object ▪ **Note** : Java.lang.Math class is final

| **Math class** |
| --- |
| import java.lang.Math;    public class MathDemo  {  public static void main(String[] args)  {  System.out.println("\n\t Square Root of 100 :" +  Math.sqrt(100));  System.out.println("\n\t Power of 2^3= " + Math.pow(2, 3));  }  } |

Program N0.

# Static Imports



* Many methods of classes like Math and Character are static.
* If the variables and methods of these classes are used very often, all are must be prefixed with Math or Character
* Normal import statement imports all classes and interfaces of a package. But static import imports only static members of a single class.
* It avoids usage of the class name multiple times in coding. The above statement avoids Math class name to prefix every variable or method used
* Ex. ceil(), floor(), pow are the static methods of Math class and PI is a static variable. All these are used without using prefix Math name.

| **Static import** |
| --- |

Program N0.

import static java.lang.Math.\*;

public class PackageDemo\_155

{

public static void main(String args[])

{

int radius = 10;

System.out.println("Perimeter: " + ceil(2 \* PI \* radius));

System.out.println("Area: " + floor(PI \* pow(radius, 2))); System.out.println("Raised 3 times: " + pow(radius, 3));

}

}

# Creating a Package



To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a package statement with that name at the top of *every source file* that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, package graphics;) must 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.

**Note:** If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define public class Circle in the file Circle.java, define public interface Draggable in the file Draggable.java, define public enum Day in the file Day.java, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be package private.

▪ **Note:**

If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

# Creating Own Package



▪ Java permits to create our own packages and use in programming

Steps of creating User Defined Packages Java and using them.

1. Create a package with a .class file
2. set the classpath from the directory from which you would like to access. It may be in a different drive and directory. Let us call it as a target directory.
3. Write a program and use the file from the package.

Let us create a package called forest and place a class called Tiger in it. Access the package from a different drive and directory.

**1st Step: Create a package (forest) and place Tiger.class in it.**

Let us assume C:\snr is the current directory where we would like to create the package.

C:\snr > notepad Tiger.java

package forest; import java.util.\*; public class Tiger

{

public void getDetails(String nickName, int weight)

{

System.out.println("Tiger nick name is " + nickName);

System.out.println("Tiger weight is " + weight); } }

When the code is ready, the next job is compilation. We must compile with package notation. Package notation uses –d compiler option as follows.

C:\snr > javac -d . Tiger.java

The **–d compiler** option creates a new folder called forest and places the Tiger.class in it. The dot (.) is an operating system's environment variable that indicates the current directory. It is an instruction to the OS to create a directory called forest and place the Tiger.class in it.

2nd step: Set the classpath from the target directory.

Let us assume D:\ javaprg is the target directory. Let us access Tiger.class in forest package from here.

From the target directory set the classpath following way.

D:\javaprg> set classpath=C:\snr;%classpath%;

**classpath** is another environment variable which gives the address of the forest directory to the

OS. **%classpath%** informs the OS to append the already existing **classpath** to the current **classpath** that is right now set.

3rd Step: Now finally, write a program from the target directory D:/sumathi and access the package.

D:\javaprg> notepad Animal.java

The above statement creates a file called Animal.java and write the code in it, say, as follows

import forest.Tiger; public class Animal

{

public static void main(String args[])

{

Tiger t1 = new Tiger (); t1.getDetails("Everest", 50);

}

}

The compilation and execution is as usual as follows.

D:\sumathi> javac Animal.java

D:\sumathi> java Animal

| **Package Student** |
| --- |
| package javaprg;    public class Student  {  public int roll; public String name; public float avg;    public Student()  {  System.out.println("\n\t Welcome in Student Class");  }    public Student(int r,String nm, float a)  {  roll = r; name= nm;  avg = a;  }    public void showStudent()  {  System.out.println("\n\t Roll Number = " + roll);  System.out.println("\n\t Name = "+ name);  System.out.println("\n\t Avg Marks = " + avg);  }  } |

**Program N0.**

# Order of Package Statement



package student; import java.util.\*; public class Student ….

* package is a keyword of Java followed by the package name. Just writing the package statement followed by the name creates a new package
* If exists, the package statement must be first one in the program
* If exists, the import statement must be the second one
* Our class declaration is the third

| **Package Example** |
| --- |
| import javaprg.\*;    class StudentDemo1  {  public static void main(String args[])  {  Student s1 = new Student();  Student s2 = new Student(101,"Niraja",45.65f);  s2.showStudent();  }  } |

Program N0.

# Mastering Javac Command



javac.exe is the Java compiler program. It compiles Java source files (.java) into bytecode class files (.class). The tool is located under JDK\_HOME\bin directory. So make sure you included this directory in the PATH environment variable so it can be accessed anywhere in command line prompt.

**Syntax of this command is:**

javac [options] [source files]

Type javac -help to view compiler options, and type javac -version to know current version of the compiler. By default, the generated .class files are placed under the same directory as the source files.

1. Compiling a single source file javac HelloWorld.java

## 2. Compiling multiple source files

1. Compile three source files at once, type:

javac Program1.java Program2.java Program3.java

1. Compile all source files whose filenames start with*Swing*

javac Swing\*.java

1. Compile all source files:

javac \*.java

## 3. Compiling a source file which has dependencies

It’s very common that a Java program depends on one or more external libraries (jar files). Use the flag -classpath (orcp) to tell the compiler where to look for external libraries (by default, the compiler is looking in bootstrap classpath and in CLASSPATH environment variable).

1. Compile a source file which depends on an external library:

javac -classpath mail.jar EmailSender.java javac -cp mail.jar EmailSender.java

1. Compile a source file which depends on multiple libraries:

javac -cp lib1.jar; lib2.jar; lib3.jar MyProgram.java javac -cp \*; MyProgram.java

## 4. Specifying destination directory

Use the -d directory option to specify where the compiler puts the generated .class files. For example:

javac -d classes MyProgram.java

NOTE:

* The compiler will complain if the specified directory does not exist, and it won’t create one.
* If the source file is under a package, the compiler will create package structure in the destination directory.

## 5. Specifying source path directory

We can tell the compiler where to search for the source files by using the-sourcepath directory option. For example

javac -sourcepath src MyProgram.java

## 6. Specifying source compatibility version

We can tell the compiler which Java version applied for the source file, by using the-source releaseoption. For example

javac -source 1.5 MyProgram.java

That will tell the compiler using specific language features in Java 1.5 to compile the source file. The valid versions are:

1.3, 1.4, 1.5 (or 5), 1.6 (or 6) and 1.7 (or 7)

# Mastering the jar tool



jar is the **Java** **archive** tool that packages (and compresses) a set of files into a single archive. The archive format is ZIP but the file name usually has .jar extension. This tool is used for creating, updating, extracting and viewing content of jar files.

The executable file of this tool can be located under the JDK\_HOME\bin directory (jar.exe on Windows), so make sure you include this path in the PATH environment variable in order to run this tool anywhere from the command line prompt.

## 1. Creating normal jar file

A normal jar file is the non-executable one, such as a library jar file or an applet jar file. The following command put all files under the build\classes directory into a new jar file called SwingEmailSender.jar: jar cfv SwingEmailSender.jar -C build\classes .

Note that there is a dot (.) at the end which denotes all files. The c option is to create, the f is to specify jar file name, the v is to generate verbose output, and the -C is to specify the directory containing the files to be added.

## 2. Including/Excluding manifest file

By default, the jar tool automatically creates a manifest file when generating a new jar file. If we don’t want to have the manifest created, use the M option as in the following example:

jar cfvM SwingEmailSender.jar -C build\classes .

In case we want to manually add an external manifest file, use the m option as in the following example:

jar cfm SwingEmailSender.jar manifest.txt -C build\classes .

Here, content of the manifest.txt is copied to the generated manifest file inside the jar file.

# Java Command



java is the Java application launcher tool which is used to execute programs written in Java programming language and compiled into bytecode class files. Its executable file can be found under JDK\_HOME\bin directory (java.exe on Windows and java on Linux), so make sure you include this path to the PATH environment variable in order to invoke the program anywhere in command line prompt.

java [options] file.class [arguments...]

java [options] -jar file.jar [arguments... ]

The first syntax is for executing a class file, and the second one is for executing a JAR file.

Type java -help to consult the available options or browse [Oracle’s Java documentation](http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html) for detailed description and explanation of the options. The arguments, if specified, will be passed into the running program.

NOTES:

* A Java class must have the public static void main(String[] args) method in order to be executed by the Java launcher.
* An executable JAR file must specify the startup class in by the Main-Class header in its manifest file

# 1. Running a Java program from a class file



Run a simple class:

If you have a source file calledMyProgram.javaand it is compiled intoMyProgram.class file, type the following command:

java MyProgram

Run a class which is declared in a package:

If the class MyProgram.java is declared in the package net.deesha, change the working directory so that it is parent of the net\deesha directory, then type java net.deesha.MyProgram

Run a class which has dependencies on jar files:

If we have a Java Mail-based program that depends onmail.jarlibrary. Assuming the jar file is at the same directory as the class file, type:

java -cp mail.jar;. PlainTextEmailSender

**NOTES:** There must be a dot (.) after the semicolon If the jar file is inside a directory called lib:

java -cp lib/mail.jar;. PlainTextEmailSender If the program depends on more than one jar files:

java -cp mail.jar;anotherlib.jar;. MyProgram

We can use wildcard character to refer to all jar files:

java -cp \*;. MyProgram

Or:

java -cp lib/\*;. MyProgram

Passing arguments to the program:

The following example passes two arguments “code” and “java” into the MyProgram:

java MyProgram code java

If the argument has spaces, we must enclose it in double quotes, for example:

java MyProgram "Welcome" 2013

That will pass two arguments “welcome” and “2013”

# 2. Running a Java program from an executable jar file



Run a standalone jar file:

java -jar MyApp.jar

Here theMyApp.jarfile must define the main class in the headerMain-Classof its manifest fileMANIFEST.MF.The header is usually created by thejartool.

**NOTE:** if the jar file depends on other jar files, the reference jar files must be specified in the headerClass-Pathof the jar’s manifest file. The-cpoption will be ignored when using -jarflag.

Passing arguments:

Pass two arguments “code” and “java” to the program:

java -jar MyApp.jar Welcome to java

# 3. Specifying splash screen



For Swing-based application, we can use the-splash:imagePathflag to show a splash screen at program’s startup. For example:

java -splash:SplashScreen.png MyProgram

Here the imageSplashScreen.pngis loaded as splash screen at startup.

# 4. Setting system properties



We can use the-Dproperty=valueoption to specify a system property when running a program:

Specify a single property:

java -Dupload.dir=D:\Uploads MyProgram

if the property’s value contains spaces, enclose it in double quotes:

java -Dupload.dir="D:\My Uploads" MyProgram

Specify multiple properties:

java -Dupload.dir=D:\Uploads -Ddownload.dir=D:\Downloads MyProgram

Override predefined property:

We can override the predefined system properties. For example, the following command overrides the system property java.io.tmpdir:

java -Djava.io.tmpdir=E:\Temp MyProgram

# 5. Specifying memory constraints



When launching a Java program, we can specify initial size and maximum size of the heap memory:

▪ -Xms<size>:specifies initial heap size ▪ -Xmx<size>: specifies maximum heap size.

The size is measured in bytes. It must be multiple of 1024 and is greater than 1MB for initial size and 2MB for maximum size. AppendkorKto indicate kilobytes;m orMto indicate megabytes. For example, the following command launches a program with initial heap size 32MB and maximum heap size 1024MB:

java -Xms32M -Xmx1024M MyProgram

# Working with Manifest Files



JAR files support a wide range of functionality, including electronic signing, version control, package sealing, and others

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

## Default Manifest

When you create a JAR file, it automatically receives a default manifest file. There can be only one manifest file in an archive, and it always has the pathname

META-INF/MANIFEST.MF

When you create a JAR file, the default manifest file simply contains the following:

Manifest-Version: 1.0

Created-By: 1.7.0\_06 (Oracle Corporation)

These lines show that a manifest's entries take the form of "header: value" pairs. The name of a header is separated from its value by a colon. The default manifest conforms to version 1.0 of the manifest specification and was created by the 1.7.0\_06 version of the JDK.

The manifest can also contain information about the other files that are packaged in the archive. Exactly what file information should be recorded in the manifest depends on how you intend to use the JAR file. The default manifest makes no assumptions about what information it should record about other files

## Setting an Application's Entry Point

If you have an application bundled in a JAR file, you need some way to indicate which class within the JAR file is your application's entry point. You provide this information with the Main-Class header in the manifest, which has the general form:

Main-Class: classname

The value classname is the name of the class that is your application's entry point.

Recall that the entry point is a class having a method with signature public static void main(String[] args).

After you have set the Main-Class header in the manifest, you then run the JAR file using the following form of the java command:

java -jar *JAR-name*

The main method of the class specified in the Main-Class header is executed.

An Example

We want to execute the main method in the class MyClass in the package MyPackage when we run the JAR file.

We first create a text file named Manifest.txt with the following contents:

Main-Class: MyPackage.MyClass

**Warning:** The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

We then create a JAR file named MyJar.jar by entering the following command:

jar cfm MyJar.jar Manifest.txt MyPackage/\*.class

This creates the JAR file with a manifest with the following contents:

Manifest-Version: 1.0

Created-By: 1.7.0\_06 (Oracle Corporation)

Main-Class: MyPackage.MyClass

When you run the JAR file with the following command, the main method of MyClass executes:

java -jar MyJar.jar

## Setting an Entry Point with the JAR Tool

The 'e' flag (for 'entrypoint') creates or overrides the manifest's Main-Class attribute. It can be used while creating or updating a JAR file. Use it to specify the application entry point without editing or creating the manifest file.

For example, this command creates app.jar where the Main-Class attribute value in the manifest is set to MyApp:

jar cfe app.jar MyApp MyApp.class

You can directly invoke this application by running the following command:

java -jar app.jar

If the entrypoint class name is in a package it may use a '.' (dot) character as the delimiter. For example, if Main.class is in a package called foo the entry point can be specified in the following ways:

jar cfe Main.jar foo.Main foo/Main.class

## Adding Classes to the JAR File's Classpath

You may need to reference classes in other JAR files from within a JAR file.

For example, in a typical situation an applet is bundled in a JAR file whose manifest references a different JAR file (or several different JAR files) that serves as utilities for the purposes of that applet.

You specify classes to include in the Class-Path header field in the manifest file of an applet or application. The ClassPath header takes the following form:

Class-Path: *jar1-name jar2-name directory-name/jar3-name*

By using the Class-Path header in the manifest, you can avoid having to specify a long -classpath flag when invoking Java to run the your application.

**Note:** The Class-Path header points to classes or JAR files on the local network, not JAR files within the JAR file or classes accessible over Internet protocols. To load classes in JAR files within a JAR file into the class path, you must write custom code to load those classes. For example, if MyJar.jar contains another JAR file called MyUtils.jar, you cannot use the Class-Path header in MyJar.jar's manifest to load classes in MyUtils.jar into the class path.

An Example

We want to load classes in MyUtils.jar into the class path for use in MyJar.jar. These two JAR files are in the same directory.

We first create a text file named Manifest.txt with the following contents:

Class-Path: MyUtils.jar

**Warning:** The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

We then create a JAR file named MyJar.jar by entering the following command:

jar cfm MyJar.jar Manifest.txt MyPackage/\*.class

This creates the JAR file with a manifest with the following contents:

Manifest-Version: 1.0

Class-Path: MyUtils.jar

Created-By: 1.7.0\_06 (Oracle Corporation)

The classes in MyUtils.jar are now loaded into the class path when you run MyJar.jar.

# Exception Handling



An exception is an abnormal event that arises during the execution of the program and disrupts the normal flow of the program. Abnormality do occur when your program is running. For example, you might expect the user to enter an integer, but receive a text string; or an unexpected I/O error pops up at runtime.

What really matters is "what happens after an abnormality occurred?" In other words, "how the abnormal situations are handled by your program." If these exceptions are not handled properly, the program terminates abruptly and may cause severe consequences. For example, the network connections, database connections and files may remain opened; database and file records may be left in an inconsistent state.

The programmer faces two types of problems in coding – problems arising at **Compile-time** and problems arising at

**Runtime**

## 1. Compile Time Errors

* All syntax errors detected and displayed by java compiler are known as Compile Time Errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary to fix all the errors before compile and run the program.
* Compile time errors occur due to typing mistakes
* Examples
  + Missing semicolons
  + Use of undeclared variables
  + Incompatible types in assignments
  + Misspelling of identifiers and keywords

## 2. Run Time Errors

▪ Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic, wrong input and may terminate. ▪ Most common Runtime Errors are :

* Dividing an integer by zero
* Accessing an element that is out of the bounds of an array - Trying to store incompatible value - Etc.

| Exception Demo |
| --- |
| import java.util.\*;  public class ExceptionDemo\_130 { |

Program N0.

|  | { | public static void main(String[] args)  Scanner scan = new Scanner(System.in); int age;  System.out.print("\n\t Enter age :"); age = scan.nextInt();  System.out.println("\n\t Age = " + age);  System.out.println("\n\t End of Program"); |
| --- | --- | --- |
| } | } | |

Java has a built-in mechanism for handling runtime errors, referred to as exception handling. This is to ensure that you can write robust programs for mission-critical applications.

Older programming languages such as C have some drawbacks in exception handing. For example, suppose the programmer wishes to open a file for processing:

1. The programmers are not made to aware of the exceptional conditions. For example, the file to be opened may not necessarily exist. The programmer therefore did not write codes to test whether the file exists before opening the file.
2. Suppose the programmer is aware of the exceptional conditions, he/she might decide to finish the main logic first, and write the exception handling codes later – this "later", unfortunately, usually never happens. In other words, you are not force to write the exception handling codes together with the main logic.
3. Suppose the programmer decided to write the exception handling codes, the exception handling codes *intertwine* with the main logic in many if-else statements. This makes main logic hard to follow and the entire program hard to read. For example,

if (file exists) { open file;

while (there is more records to be processed) { if (no IO errors) { process the file record

} else {

handle the errors

} } if (file is opened) close the file;

} else {

report the file does not exist; }

Java overcomes these drawbacks by building the exception handling into the language rather than leaving it to the discretion of the programmers:

1. You will be informed of the exceptional conditions that may arise in calling a method - Exceptions are declared in the method's signature.
2. You are forced to handle exceptions while writing the main logic and cannot leave them as an afterthought - Your program cannot compiled without the exception handling codes.
3. Exception handling codes are separated from the main logic - Via the try-catch-finally construct.

# Exception



▪ An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

* A user has entered 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.

# How Exception Handling



The purpose of exception handling mechanism is to provide a system to detect and report an “Exceptional circumstance” so that appropriate actions can be taken.

▪ Exception handling mechanism have following tasks

1. Find the Problem **(Hit the Exception)**
2. Inform that an error has occurred **(Throw the Exception)**
3. Receive the Error information **(Catch the Exception)**
4. Take corrective actions **(Handle the Exception)**

# How to Handle Exception



Java uses try block that contains one or more statements that could generate an exception.

Syntax

**try**

**{**

**//statements that cause Exception**

**}**

catch(ExceptionType obj)

{ //statements that handle Exception }

* If any statement generates an Exception, the remaining statements in the block are skipped and execution jumps to the catch block.
* The catch block “catches” the exception “thrown” by the try block and handles it appropriately.
* The catch statement is passed a single parameter, which is reference to the Exception object thrown.
* If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed.

| Exception Demo Arithmetic Exception |
| --- |
| import java.util.\*;    public class ExceptionDemo\_131  { public static void main(String[] args)  {  Scanner scan = new Scanner(System.in); int age;    try  {  System.out.print("\n\t Enter age :");  age = scan.nextInt();    System.out.println("\n\t Age = " + age); |

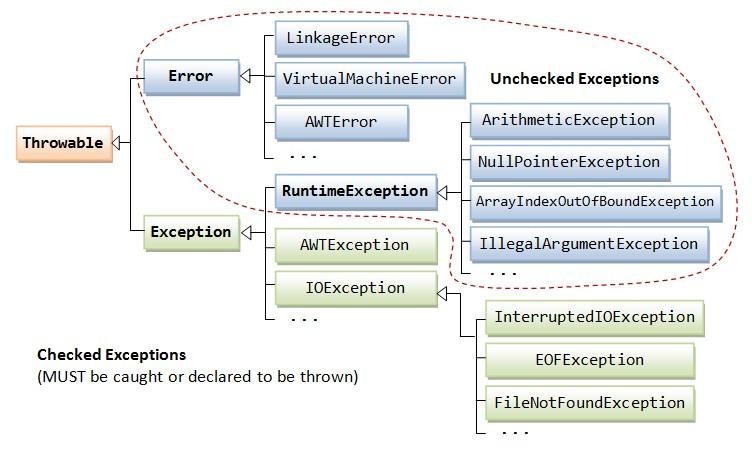
Program N0. 131

| } | }  catch(InputMismatchException e)  {  System.out.println("\n\t Exception Caught");  System.out.println("\n\t Error : " + e); }  System.out.println("\n\t End of Program"); } |
| --- | --- |

# Exception Classes



The figure below shows the hierarchy of the Exception classes. The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception andjava.lang.Error.



The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.

The Exception class describes the error caused by your program (e.g. FileNotFoundException, IOException). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw just any object as an exception, however -- only those objects whose classes descend from Throwable. Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw.

Throwable has two direct subclasses, Exception and Error. Exceptions (members of the Exception family) are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread. Errors (members of the Error family) are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle. In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

In addition to throwing objects whose classes are declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of

the Throwable family. In general, however, the throwable classes you define should extend class Exception. They should be "exceptions."

Whether you use an existing exception class from java.lang or create one of your own depends upon the situation. In some cases, a class from java.lang will do just fine. For example, if one of your methods is invoked with an invalid argument, you could throw IllegalArgumentException, a subclass of RuntimeException in java.lang.

## Advantages of Exceptions



Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

### Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

readFile { open the file; determine its size; allocate that much memory; read the file into memory; close the file;

}

At first glance, this function seems simple enough, but it ignores all the following potential errors.

1. What happens if the file can't be opened?
2. What happens if the length of the file can't be determined?
3. What happens if enough memory can't be allocated?
4. What happens if the read fails?
5. What happens if the file can't be closed?

To handle such cases, the readFile function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

errorCodeType readFile { initialize errorCode = 0;

open the file; if (theFileIsOpen) {

determine the length of the file; if (gotTheFileLength) { allocate that much memory; if (gotEnoughMemory) { read the file into memory; if (readFailed) { errorCode = -1;

}

} else { errorCode = -2;

} } else { errorCode = -3;

}

close the file;

if (theFileDidntClose && errorCode == 0) { errorCode = -4;

} else {

errorCode = errorCode and -4;

} } else { errorCode = -5;

}

return errorCode;

}

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the readFile function used exceptions instead of traditional error-management techniques, it would look more like the following.

readFile { try {

open the file; determine its size; allocate that much memory; read the file into memory; close the file; } catch (fileOpenFailed) { doSomething;

} catch (sizeDeterminationFailed) { doSomething;

} catch (memoryAllocationFailed) { doSomething;

} catch (readFailed) {

doSomething;

} catch (fileCloseFailed) { doSomething;

}

}

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

### Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

method1 { call method2;

} method2 { call method3;

} method3 {

call readFile; }

Suppose also that method1 is the only method interested in the errors that might occur within readFile. Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

method1 {

errorCodeType error; error = call method2; if (error) doErrorProcessing; else proceed;

}

errorCodeType method2 { errorCodeType error; error = call method3; if (error) return error; else proceed;

}

errorCodeType method3 { errorCodeType error; error = call readFile; if (error) return error; else proceed;

}

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

method1 { try { call method2; } catch (exception e) { doErrorProcessing;

} }

method2 throws exception {

call method3;

}

method3 throws exception { call readFile; }

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its throws clause.

### Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in java.io —IOException and its descendants. IOException is the most general and represents any type

of error that can occur when performing I/O. Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception.

catch (FileNotFoundException e) { ...

}

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument.

catch (IOException e) { ...

}

This handler will be able to catch all I/O exceptions, including FileNotFoundException, EOFException, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

catch (IOException e) {

// Output goes to System.err.

e.printStackTrace(); // Send trace to stdout.

e.printStackTrace(System.out); }

You could even set up an exception handler that handles any Exception with the handler here.

// A (too) general exception handler catch (Exception e) { ...

}

The Exception class is close to the top of the Throwable class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

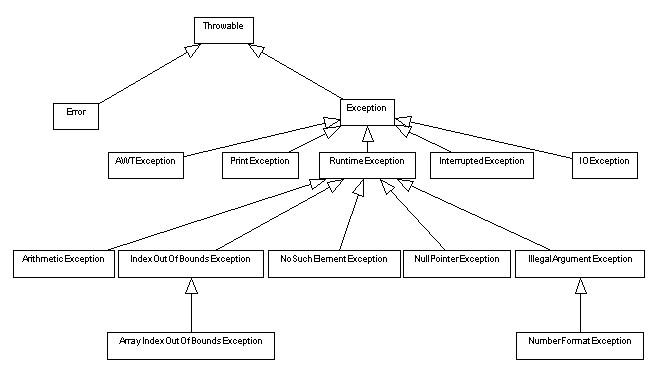
As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

## Checked vs. Unchecked Exceptions



A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. These exceptions cannot simply be ignored at the time of compilation. These exceptions are explicitly handled in code itself with the help of try – catch blocks

Unchecked Exceptions are not essentially handled in the program code, instead the JVM handles such Exceptions. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.



As illustrated, the subclasses of Error and RuntimeException are known as unchecked exceptions. These exceptions are not checked by the compiler, and hence, need not be caught or declared to be thrown in your program.

This is because there is not much you can do with these exceptions. For example, a "divide by 0" triggers an ArithmeticException, array index out-of-bound triggers an ArrayIndexOutOfBoundException, which are really programming logical errors that shall be been fixed in compiled-time, rather than leaving it to runtime exception handling.

All the other exceptions are called checked exceptions. They are checked by the compiler and must be caught or declared to be thrown.

## Printing Exception



### System.out.println(ex)

Prints the string representation of exception object. Prints the exception class name (java.lang.ArithmeticException) and also the exception message

System.out.println(ex.getMessage())

Prints exception message (Ex. / by zero) only indicating the cause of exception

### e.printStackTrace()

Prints the exception class name with the message particulars and also the line number where the problem arises (traces the actual problem).

## Common Exception Classes



### NumberFormatException

* It is an unchecked exception thrown by **parseXXX()** methods when they are unable to format (convert) a string into a number.
* Sometimes, in Java coding, we get input (like from command-line arguments and text field) from the user in the form of string. To use the string in arithmetic operations, it must be converted (parsed) into data types. This is done by parseXXX() methods of wrapper classes.
* Object –> Throwable –> Exception –> RuntimeException –> NumberFormatException

| **Exception Demo** |
| --- |
| public class ExceptionDemo  {  public static void main(String[] args)  {  String str = "onetwothree";  int no;    try  {  no = Integer.parseInt(str);  System.out.println("\n\t no = " + no);  }  catch(NumberFormatException e)  {  System.out.println("\n\t Error :" + e.getMessage());  }  System.out.println("\n\t End of Program");  }  } |

Program N0.

### ArrayIndexOutOfBoundsException

Thrown by JVM when your code uses an array index, which is is outside the array's bounds. For example,

int[] anArray = new int[3];

System.out.println(anArray[3]);

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3

### NullPointerException

Thrown by the JVM when your code attempts to use a null reference where an object reference is required. For example,

String[] strs = new String[3];

System.out.println(strs[0].length());

Exception in thread "main" java.lang.NullPointerException

### ClassCastException

Thrown by JVM when an attempt is made to cast an object reference fails. For example,

Object o = new Object();

Integer i = (Integer)o;

Exception in thread "main" java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.Integer

### IllegalArgumentException

Thrown programmatically to indicate that a method has been passed an illegal or inappropriate argument. You could reuse this exception for your own methods.

### IllegalStateException

Thrown programmatically when a method is invoked and the program is not in an appropriate state for that method to perform its task. This typically happens when a method is invoked out of sequence, or perhaps a method is only allowed to be invoked once and an attempt is made to invoke it again.

### NoClassDefFoundError

Thrown by the JVM or class loader when the definition of a class cannot be found. Prior to JDK 1.7, you will see this exception call stack trace if you try to run a non-existent class. JDK 1.7 simplifies the error message to "Error: Could not find or load main class xxx".

## Multiple Catch Blocks



It is possible to have more than one catch statement in the try-catch block

| **Exception Demo** |
| --- |
| import java.util.\*;    public class ExceptionDemo  { public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int n1, n2, ans;    try  {  System.out.print("\n\t Enter Number 1 :"); n1 = scan.nextInt(); |

Program N0.

| } | System.out.print("\n\t Enter Number 2 :");  n2 = scan.nextInt();  ans = n1/n2;  System.out.println("\n\t Answer = " + ans);  }  catch(InputMismatchException e)  {  System.out.println("\n\t Exception Caught"); e.printStackTrace();  }  catch(ArithmeticException e)  {  System.out.println("\n\t Can not / by zero");  }  catch(Exception e)  {  }  System.out.println("\n\t End of Program"); } |
| --- | --- |

When an exception in the try block is generated, the Java treats the multiple catch Statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

**Note:** Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion

## Handling exceptions – Three styles



1. Using try-catch block; the robust way
2. Using throws in place of try-catch, not a robust way
3. To throw the exception object to the system using throw keyword, not a robust way

## Exception Handling Operations



Five keywords are used in exception handling: try, catch, finally, throws and throw (take note that there is a difference between throw and throws).

Java’s exception handling consists of three operations:

1. Declaring exceptions
2. Throwing an exception
3. Catching an exception

### Declaring Exceptions

A Java method must declare in its signature the types of checked exception it may "throw" from its body, via the keyword "throws".

For example, suppose that methodD() is defined as follows:

public void methodD() throws XxxException, YyyException {

// method body throw XxxException and YyyException }

The method's signature indicates that running methodD() may encounter two checked exceptions: XxxException and YyyException. In other words, some of the abnormal conditions insidemethodD() may trigger XxxException or YyyException.

Exceptions belonging to Error, RuntimeException and their *subclasses* need not be declared. These exceptions are called *unchecked exceptions* because they are not checked by the compiler.

### Throwing an Exception

When a Java operation encounters an abnormal situation, the method containing the erroneous statement shall create an appropriate Exception object and throw it to the Java runtime via the statement "throw XxxException". For example, public void methodD() throws XxxException, YyyException { // method's signature

// method's body

...

...

// XxxException occurs if ( ... ) throw new XxxException(...); // construct an XxxException object and throw

to JVM

...

// YyyException occurs if ( ... )

throw new YyyException(...); // construct an YyyException object and throw

to JVM ...

}

Note that the keyword to declare exception in the method's signature is "throws" and the keyword to throw an exception object within the method's body is "throw".

### Catching an Exception

When a method throws an exception, the JVM searches backward through the call stack for a matching exception handler. Each exception handler can handle one particular class of exception. An exception handler handles a specific class can also handle its subclasses. If no exception handler is found in the call stack, the program terminates.

For example, suppose methodD() declares that it may throw XxxException and YyyException in its signature, as follows:

public void methodD() throws XxxException, YyyException { ...... }

To use methodD() in your program (says in methodC()), you can either:

1. Wrap the call of methodD() inside a try-catch (or try-catch-finally) as follows. Each catch-block can contain an exception handler for one type of exception.

public void methodC() { // no exception declared ...... try { ......

// uses methodD() which declares XxxException & YyyException methodD();

......

} catch (XxxException ex) {

// Exception handler for XxxException

......

} catch (YyyException ex} {

// Exception handler for YyyException

......

} finally { // optional

// These codes always run, used for cleaning up ......

} ...... }

2. Suppose that methodC() who calls methodD() does not wish to handle the exceptions (via a try-catch), it can declare these exceptions to be thrown up the call stack in its signature as follows:

public void methodC() throws XxxException, YyyException { // for next higher-level method to handle

...

// uses methodD() which declares "throws XxxException, YyyException" methodD(); // no need for try-catch ... }

In this case, if a XxxException or YyyException is thrown by methodD(), JVM will *terminate* methodD() as well as methodC() and pass the exception object up the call stack to the caller of methodC().

## Point 1: Exceptions must be declared



As an example, suppose that you want to use a java.util.Scanner to perform formatted input from a disk file. The signature of the Scanner's constructor with a File argument is given as follows:

public Scanner(File source) throws FileNotFoundException;

The method's signature informs the programmers that an exceptional condition "file not found" may arise. By declaring the exceptions in the method's signature, programmers are made to aware of the exceptional conditions in using the method.

## Point 2: Exceptions must be handled



If a method declares an exception in its signature, you cannot use this method without handling the exception - you can't compile the program.

**Example 1:** The program did not handle the exception declared, resutled in compilation error.

import java.util.Scanner; import java.io.File; public class ScannerFromFile {

public static void main(String[] args) { Scanner in = new Scanner(new File("test.in")); // do something ...

}

}

**ERROR**

ScannerFromFile.java:5: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown

**Scanner in = new Scanner(new File("test.in"));**

To use a method that declares an exception in its signature, you MUST either:

1. provide exception handling codes in a "try-catch" or "try-catch-finally" construct, or
2. if don’t want handling the exception in the current method, but declare the exception to be thrown up the call stack for the next higher-level method to handle.

**Example 2:** Catch the exception via a "**try-catch**" (or "**try-catch-finally**") construct.

import java.util.Scanner; import java.io.File;

import java.io.FileNotFoundException; public class ScannerFromFileWithCatch { public static void main(String[] args) { try {

Scanner in = new Scanner(new File("test.in"));

// do something if no exception ...

// you main logic here in the try-block

}

catch (FileNotFoundException ex)

{

// error handling separated from the main logic

ex.printStackTrace(); // print the stack trace

}

} }

If the file cannot be found, the exception is caught in the catch-block. In this example, the error handler simply prints the *stack trace*, which provides useful information for debugging. In some situations, you may need to perform some clean-up operations, or open another file instead. Take note that the main logic in the try-block is separated from the error handling codes in the catch-block.

**Example 3:** You decided not to handle the exception in the current method, but throw the exception up the call stack for the next higher-level method to handle.

import java.util.Scanner; import java.io.File;

import java.io.FileNotFoundException; public class ScannerFromFileWithThrow {

public static void main(String[] args) throws FileNotFoundException { // to be handled by next higher-level method

Scanner in = new Scanner(new File("test.in"));

// this method may throw FileNotFoundException // main logic here ...

} }

In this example, you decided not to handle the FileNotFoundException thrown by the Scanner(File) method

(with try-catch). Instead, the caller of Scanner(File) - the main() method - declares in its signature "throws

FileNotFoundException", which means that this exce ption will be thrown up the call stack, for the next higher-level method to handle. In this case, the next higher-level method of main() is the JVM, which simply terminates the program and prints the stack trace.

## Point 3: Main logic is separated from the exception handling codes



As shown in Example 2, the main logic is contained in the try-block, while the exception handling codes are kept in the catch-block(s) separated from the main logic. This greatly improves the readability of the program.

For example, a Java program for file processing could be as follows:

try {

// Main logic here open file; process file; ......

} catch (FileNotFoundException ex) { // Exception handlers below

// Exception handler for "file not found"

} catch (IOException ex) {

// Exception handler for "IO errors"

} finally {

close file; // always try to close the file }

## Throws Keyword



### What if I really don't care about the exceptions

Certainly not advisable other than writing toy programs. But to bypass the compilation error messages triggered by methods declaring unchecked exceptions, you could declare "throws Exception" in your main() (and other methods), as follows:

public static void main(String[] args) **throws Exception** { // throws all subclass of Exception to JRE

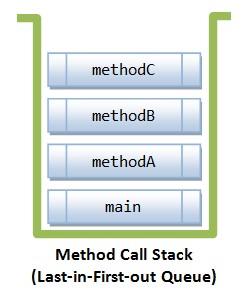
Scanner in = new Scanner(new File("test.in")); // declares "throws FileNotFoundException" ......

// other exceptions

}

## Method Call Stack



A typical application involves many levels of method calls, which is managed by a so-called *method call stack*. A *stack* is a last-in-first-out queue. In the following example, main() method invokes methodA(); methodA() calls methodB(); methodB() calls methodC(). 

public class MethodCallStackDemo { public static void main(String[] args) { System.out.println("Enter main()"); methodA();

System.out.println("Exit main()");

}

public static void methodA() {

System.out.println("Enter methodA()"); methodB();

System.out.println("Exit methodA()");

}

public static void methodB() {

System.out.println("Enter methodB()"); methodC();

System.out.println("Exit methodB()");

}

public static void methodC() {

System.out.println("Enter methodC()");

System.out.println("Exit methodC()");

}

}

Enter main()

Enter methodA()

Enter methodB()

Enter methodC()

Exit methodC()

Exit methodB()

Exit methodA()

Exit main()

As seen from the output, the sequence of events is:

1. JVM invoke the main().
2. main() pushed onto call stack, before invoking methodA().
3. methodA() pushed onto call stack, before invoking methodB().
4. methodB() pushed onto call stack, before invoking methodC().
5. methodC() completes.
6. methodB() popped out from call stack and completes.
7. methodA() popped out from the call stack and completes.
8. main() popped out from the call stack and completes. Program exits.

Suppose that we modify methodC() to carry out a "divide-by-0" operation, which triggers a ArithmeticException:

public static void methodC() {

System.out.println("Enter methodC()");

System.out.println(1 / 0); // divide-by-0 triggers an ArithmeticException

System.out.println("Exit methodC()"); }

The exception message clearly shows the *method call stack trace* with the relevant statement line numbers:

Enter main()

Enter methodA()

Enter methodB()

Enter methodC()

Exception in thread "main" java.lang.ArithmeticException: / by zero at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22) at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16) at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10) at MethodCallStackDemo.main(MethodCallStackDemo.java:4)

MethodC() triggers an ArithmeticException. As it does not handle this exception, it popped off from the call stack immediately. MethodB() also does not handle this exception and popped off the call stack.

So does methodA() and main() method. The main() method passes back to JVM, which abruptly terminates the program and print the call stack trace, as shown.

## Exception & Call Stack



When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JVM (in Java term, the method "throw" an Exception).

The Exceptionobject contains the type of the exception, and the state of the program when the exception occurs. The JVM is responsible for finding an exception handler to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception). If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

This process is illustrated as follows. Suppose that methodD() encounters an abnormal condition and throws a XxxException to the JVM. The JVM searches backward through the call stack for a matching exception handler. It finds methodA() having a XxxException handler and passes the exception object to the handler. Notice

that methodC() and methodB() are required to declare "throws XxxException" in their method signatures in order to compile the program.

## try-with-resources structure



So far you have been familiar with the try-catch-finally structure. Now I’m about to tell you the advanced version of exception handling in Java - it is the **try-with-resources** structure which was added to the Java language from Java SE 7.

Let’s look at a typical try-catch-finally example I showed you previously:

FileWriter writer = null; try {

writer = new FileWriter("Name.txt"); writer.write("Hello "); writer.close(); } catch (IOException e) {

e.printStackTrace(); } finally {

if (writer != null) { try {

writer.close();

} catch (IOException ce) { ce.printStackTrace();

}

}

}

The finally block is usually used to close a resource such as a file, a network connection, a database connection and the like. This pattern is repeated again and again so Java 7 makes our lives easier by enhancing the exception handling with the introduction of try-with-resources structure.

The above code can be re-written using the try-with-resources construct as follows:

try (FileWriter writer = new FileWriter("Name.txt")) { writer.write("Hello "); writer.close(); } catch (IOException e) {

e.printStackTrace(); }

With this new structure, we don’t have to explicitly close the resource used by the finally block. Instead, the Java compiler will figure it out and automatically adds code to close the resource for us.

Well, the secret lies in the **AutoCloseable** interface that defines only a single method:

public void close();

So when a resource used by the try block implements this interface, the compiler knows that it’s safe to call the **close()** method on the resource object.

That means the try-with-resources structure works only with AutoCloseable’s implementations. And fortunately, Java 7 refactors almost resource-like classes to implement this interface to support programmers.

Thanks to the try-with-resources construct that brings us the following benefits:

* We can write more compact code as eliminating the finally block. This saves time.
* We can write more safe and efficient code as if we forget to close a resource, the compiler does the work for us behind the scenes. Using a database connection within a **try-catch-finally** structure:

Connection conn = null;

try {

String dbURL = "jdbc:oracle:thin:tiger/scott@localhost:1521:DB"; conn = DriverManager.getConnection(dbURL);

// execute SQL statements } catch (SQLException ex) { ex.printStackTrace();

} finally { try {

if (conn != null && !conn.isClosed()) { conn.close();

}

} catch (SQLException ex) { ex.printStackTrace();

}

}

It is now more compact with try-with-resources version:

String dbURL = "jdbc:oracle:thin:tiger/scott@localhost:1521:DB";

try (Connection conn = DriverManager.getConnection(dbURL)) {

// execute SQL statements } catch (SQLException ex) { ex.printStackTrace();

}

**NOTE:**we can use initialize multiple resources in the try block and the compiler is smart enough to close them all. Here’s an example that copies one file to another using the try-catch-finally fashion:

public void copyFile(File sourceFile, File destFile)

throwsIOException {

FileChannel sourceChannel = null; FileChannel destChannel = null; try{

sourceChannel = newFileInputStream(sourceFile).getChannel(); destChannel = newFileOutputStream(destFile).getChannel(); sourceChannel.transferTo(0, sourceChannel.size(), destChannel);

} finally{

if(sourceChannel != null) { sourceChannel.close();

}

if(destChannel != null) { destChannel.close();

}

}

}

And now with the try-with-resources fashion:

public void copyFile(File sourceFile, File destFile) throwsIOException {

try(

FileChannel sourceChannel = newFileInputStream(sourceFile).getChannel(); FileChannel destChannel = newFileOutputStream(destFile).getChannel();

) {

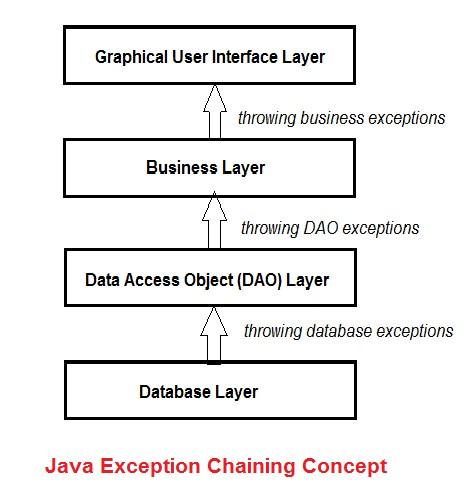
sourceChannel.transferTo(0, sourceChannel.size(), destChannel); }

}

Exception Chaining?



Basically, exception chaining is the process of re-throwing multiple exceptions across different abstraction layers of a program. The key principle here is that, these exceptions are chained together to maintain the stack trace from the exception at the lowest layer to the one at the highest layer. The following picture illustrates this concept visually:



As you can see, each abstraction layer defines its own exception classes. When code in a layer throws an exception, the higher layer re-throws it under a new type of exception which corresponds to the abstraction level of that layer. In turn, the next higher layer re-throws the exception under its own type of exception, and so on. This process continues until a layer handles the exception instead of re-throwing. During this chaining process, the higher exception always wraps the lower exception as its cause. Therefore, when an exception occurs, the programmer has a complete stack trace of the exceptions, which is very helpful for debugging.

* **Why is Exception Chaining?**

The main purpose of exception chaining is to preserve the original exception when it propagates across multiple logical layers in a program. This is very helpful for the debugging process when an exception is thrown, as the programmer can analyze the full stack trace of the exceptions.

In addition, exception chaining also helps promoting abstraction among logical layers in a program, as each layer defines its own exceptions which are specific for that layer. For example, the StudentBusinessclass throws StudentException would be more meaningful than SQLException, right?

You know, exception chaining is sometimes referred as *exception propagation*, as when a layer throws an exception, the exception propagates through higher layers until a layer handles it such as displaying a message/warning to the user.

* **How to Chain Exceptions Together?**

Let’s consider the following code example:

public void setBirthday(String birthDate) throws InvalidBirthdayException {

DateFormat formatter = new SimpleDateFormat(); try {

Date birthday = formatter.parse(birthDate);

} catch (ParseException ex) {

throw new InvalidBirthdayException("Date of birth is invalid", ex);

}

}

As you can see in the setBirthday() method, the ParseException is re-thrown under a new exception called InvalidBirthdayException. The ParseException is chained via the constructor of InvalidBirthdayException class:

throw new InvalidBirthdayException("Date of birth is invalid", ex);

This custom exception is implemented as following:

public class InvalidBirthdayException extends Exception {

public InvalidBirthdayException(String message, Throwable cause) { super(message, cause);

}

}

You can notice that, this constructor invokes its super’s constructor:

super(message, cause);

The supertypes of all exceptions Throwable and Exception implement this constructor, so any custom exceptions can call it. The origin exception (the cause) is passed to the being-created exception via its constructor.

Remember that the Exception class provides the following constructors that help chaining an exception:

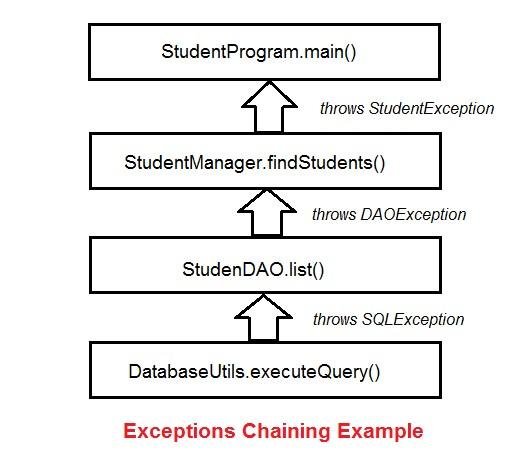
* Exception(Throwable cause)
* Exception(String message, Throwable cause)

Besides chaining an exception via constructor, you can also chain an exception through the following Throwable’s method:

public Throwable initCause(Throwable cause)

That’s how exceptions are chained together.

Let’s see another example which is illustrated by the following picture:



And following is source code of each class.

**DAOException.java:**

public class DAOException extends Exception { public DAOException(String message, Throwable cause) { super(message, cause);

}

}

**StudentException.java:**

public class StudentException extends Exception { public StudentException(String message) { super(message);

}

public StudentException(String message, Throwable cause) { super(message, cause);

}

}

**DatabaseUtils.java:**

import java.sql.\*; public class DatabaseUtils {

public static void executeQuery(String sql) throws SQLException { throw new SQLException("Syntax Error");

}

}

**StudentDAO.java:**

import java.sql.\*; public class StudentDAO {

public void list() throws DAOException {

try {

DatabaseUtils.executeQuery("SELECT");

} catch (SQLException ex) {

throw new DAOException("Error querying students from database", ex);

}

}

}

**StudentManager.java:**

public class StudentManager { private StudentDAO dao;

public StudentManager(StudentDAO dao) { this.dao = dao;

}

public void findStudents(String keyword) throws StudentException { try {

dao.list();

} catch (DAOException ex) {

throw new StudentException("Error finding students", ex);

}

}

}

**StudentProgram.java:**

public class StudentProgram {

public static void main(String[] args) { StudentDAO dao = new StudentDAO();

StudentManager manager = new StudentManager(dao); try {

manager.findStudents("Tom"); } catch (StudentException ex) { ex.printStackTrace();

}

}

}

Run the StudentProgram and you should see the following output:

StudentException: Error finding students

at StudentManager.findStudents(StudentManager.java:13) at StudentProgram.main(StudentProgram.java:9)

Caused by: DAOException: Error querying students from database at StudentDAO.list(StudentDAO.java:11)

at StudentManager.findStudents(StudentManager.java:11)

... 1 more

Caused by: java.sql.SQLException: Syntax Error at DatabaseUtils.executeQuery(DatabaseUtils.java:5) at StudentDAO.list(StudentDAO.java:8)

... 2 more

You see? The printed exception stack trace reveals an exception propagates from the DatabaseUtils layer up to the StudentProgram layer in which the exception is handled by printing this trace.

## try-catch-finally



The syntax of try-catch-finally is: **try {**

// main logic, uses methods that may throw Exceptions ......

**} catch (***Exception1 ex***) {**

// error handler for Exception1 ......

**} catch (***Exception2 ex***) {**

// error handler for Exception1 ......

**} finally {** // finally is optional

// clean up codes, always executed regardless of exceptions ......

**}**

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks sequentially.

If a catch-block catches that exception class or catches a *superclass* of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception.

The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a System.exit() is called in the catch block).

Example 1 import java.util.Scanner; import java.io.File;

import java.io.FileNotFoundException; public class TryCatchFinally {

public static void main(String[] args) {

try { // main logic

System.out.println("Start of the main logic"); System.out.println("Try opening a file ...");

Scanner in = new Scanner(new File("test.in"));

System.out.println("File Found, processing the file ...");

System.out.println("End of the main logic");

} catch (FileNotFoundException ex) { // error handling separated from the main logic

System.out.println("File Not Found caught ...");

} finally { // always run regardless of exception status

System.out.println("finally-block runs regardless of the state of exception");

}

// after the try-catch-finally

System.out.println("After try-catch-finally, life goes on...");

}

}

This is the output when the FileNotFoundException triggered: Start of the main logic Try opening a file ... File Not Found caught ... finally-block runs regardless of the state of exception After try-catch-finally, life goes on...

This is the output when no exception triggered:

Start of the main logic Try opening a file ...

File Found, processing the file ... End of the main logic

finally-block runs regardless of the state of exception After try-catch-finally, life goes on..

### try-catch-finally

1. A try-block must be accompanied by at least one catch-block or a finally-block.
2. You can have multiple catch-blocks. Each catch-block catches only one type of exception.
3. A catch block requires one argument, which is a throwable object (i.e., a subclass of java.lang.Throwable), as follows:

catch (AThrowableSubClass aThrowableObject) {

// exception handling codes

}

You can use the following methods to retrieve the type of the exception and the state of the program from the Throwable object:

**printStackTrace ():** Prints this Throwable and its call stack trace to the standard error stream System.err. The first line of the outputs contains the result of toString(), and the remaining lines are the stack trace. This is the most common handler, if there is nothing better that you can do. For example,

try {

Scanner in = new Scanner(new File("test.in"));

// process the file here

......

} catch (FileNotFoundException ex) {

**ex.printStackTrace();**

}

You can also use printStackTrace(PrintStream s) or printStackTrace(PrintWriter s).

* **getMessage():** Returns the message specified if the object is constructed using constructor Throwable(String

message).

* **toString():** Returns a short description of this Throwable object, consists of the name of the class, a colon ':', and a message from getMessage().

* A catch block catching a specific exception class can also catch its *subclasses*. Hence, catch(Exception ex) {...} catches all kinds of exceptions. However, this is not a good practice as the exception handler that is too general may unintentionally catches some subclasses' exceptions it does not intend to.

* The order of catch-blocks is important. A subclass must be caught (and placed in front) before its superclass. Otherwise, you receive a compilation error "exception XxxException has already been caught".

* The finally-block is meant for cleanup code such as closing the file, database connection regardless of whether the try block succeeds. The finally block is always executed (unless the catch-block pre-maturely terminated the current method).

| Finally Block |
| --- |
| public class ExceptionDemo  {  public static void main(String[] args)  {  String str = "onetwothree";  int no;    try  {  no = Integer.parseInt(str);  System.out.println("\n\t no = " + no);  }  catch(NumberFormatException e)  {  System.out.println("\n\t Error :" +  e.getMessage());  } |

Program N0.

|  |  | finally  {  System.out.println("\n\t End of Program"); } |
| --- | --- | --- |
| } | } |  |

## Overriding and Overloading Methods



An overriding method must have the same argument list and return-type (or subclass of its original from JDK 1.5). An overloading method must have different argument list, but it can have any return-type.

An overriding method cannot have more restricted access. For example, a method with protected access may be overridden to have protected or public access but not private or default access. This is because an overridden method is considered to be a replacement of its original, hence, it cannot be more restrictive.

An overriding method cannot declare exception types that were not declared in its original. However, it may declare exception types are the same as, or subclass of its original. It needs not declare all the exceptions as its original. It can throw fewer exceptions than the original, but not more.

An overloading method must be differentiated by its argument list. It cannot be differentiated by the return-type, the exceptions, and the modifier, which is illegal. It can have any return-type, access modifier, and exceptions, as long as it can be differentiated by the argument list.

## Creating Your Own Exception Classes



You should try to reuse the Exception classes provided in the JDK,

e.g., IndexOutOfBoundException, ArithmeticException, IOException, and IllegalArugmentException. But you can always create you own Exception classes by extending from the class Exception or one of its subclasses.

Note that RuntimeException and its subclasses are not checked by the compiler and need not be declared in the method's signature. Therefore, use them with care, as you will not be informed and may not be aware of the exceptions that may occur by using that method (and therefore do not have the proper exception handling codes) – a bad software engineering practice.

Example

// Create our own exception class by subclassing Exception. This is a checked exception public class MyMagicException extends Exception { public MyMagicException(String message) { //constructor super(message);

} }

public class MyMagicExceptionTest {

// This method "throw MyMagicException" in its body.

// MyMagicException is checked and need to be declared in the method's signature

public static void magic(int number) throws MyMagicException { if (number == 8) {

throw (new MyMagicException("you hit the magic number"));

}

System.out.println("hello"); // skip if exception triggered

}

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

magic(9); // does not trigger exception magic(8); // trigger exception

} catch (MyMagicException ex) { // exception handler ex.printStackTrace();

}

}

}

The output is as follows:

hello

MyMagicException: you hit the magic number

at MyMagicExceptionTest.magic(MyMagicExceptionTest.java:6) at MyMagicExceptionTest.main(MyMagicExceptionTest.java:14)

| **User Defined Exception** |
| --- |
| import java.util.\*;    class InvalidNo extends Exception  {  public InvalidNo(String msg)  {  super(msg);  }  }  public class ExceptionDemo  {  public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);    try  {  int no;  System.out.print("\n\t Enter number :");  no = scan.nextInt();    if(no>=0 && no<=9)  {  System.out.println("\n\t No = " + no);  }  else  {  throw new InvalidNo("No is Not Valid");  }  }  catch(InvalidNo e)  {  System.out.println("\n\t Error = " + e.getMessage());  }  catch(InputMismatchException e)  {  System.out.println("\n\t Error = " + e);  }  }  } |

Program N0.

## Input & Output (I/O) java.io Package



Programs read inputs from data sources (e.g., keyboard, file, network, memory buffer, or another program) and write outputs to data sinks (e.g., display console, file, network, memory buffer, or another program).

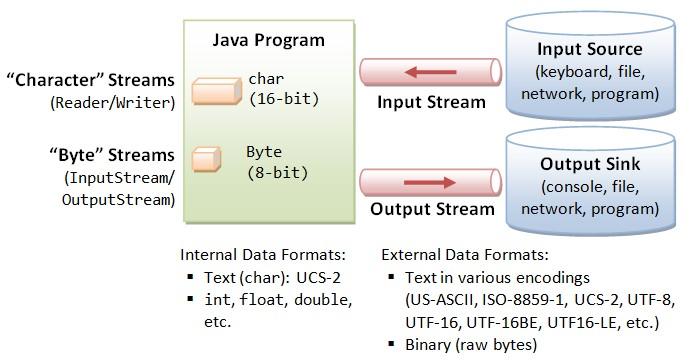
In Java standard I/O, inputs and outputs are handled by the so-called **streams**. A stream is a sequential and contiguous one-way flow of data (just like water or oil flows through the pipe). It is important to mention that Java does not differentiate between the various types of data sources or sinks (e.g., file or network) in stream I/O. They are all treated as a sequential flow of data.

Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console or another program. The Java program receives data from a source by opening an input stream, and sends data to a sink by opening an output stream.

**Stream I/O operations involve three steps:**

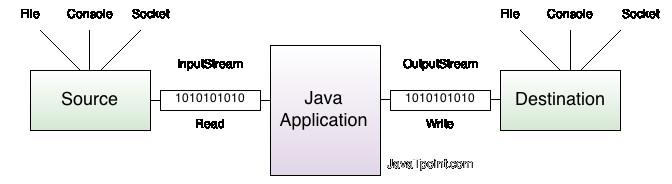
1. Open an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
2. Read from the opened input stream until "end-of-stream" encountered, or *write* to the opened output stream (and optionally flush the buffered output).
3. Close the input/output stream.

As a consequence, Java needs to differentiate between byte-based I/O for processing raw bytes or binarydata, and character-based I/O for processing texts made up of characters.



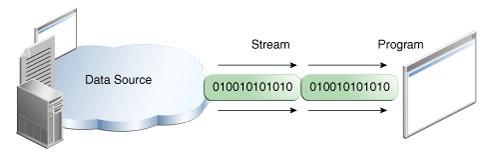
A Stream presents a uniform, easy-to-use, object oriented interface between the program and input/output devices. A Stream in java is a path along which data flows. It has a source (of data) and a destination (for that data). Both the source and destination may a physical devices or programs or other streams in the same program

The term stream refers to a sequence of values from any input source of data (keyboard, file, port, and so on), or to any output destination for data (screen, file, port, and so on). The actual physical source or destination must be supplied as an argument to an appropriate constructor of an appropriate class, during the construction of the actual object that will do the physical reading or writing of data.



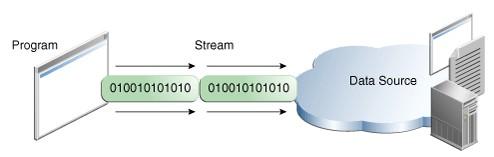
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: **A stream is a sequence of data**. A program uses an *input stream* to read data from a source, one item at a time:



Reading information into a program.

A program uses an output stream to write data to a destination, one item at time:



Writing information from a program.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

## I/O Streams



The classes in java.io package may be categorized into two groups based on the data type on which they operate 

* Byte Stream classes – provides support for handling I/O operations on bytes
* Character stream Class - provides support for managing I/O operations on characters

These two groups may be further classified bases based on their purpose.

Byte Streams and Character Stream classes contains specialized classes to deal with input and output operations on various types of devices

## Keyboard Input Using DataInputStream



DataInputStream dis = new DataInputStream(System.in);

What is "System.in"?

" **in**" is an object of " **InputStream**" class defined in System class (like "out" is an object of PrintStream class defined in System class). It is declared as static and final object. The **in** object is connected implicitly to the " **standard input stream**" of the underlying OS

| **Keyboard Input using DataInputStream** |
| --- |
| import java.io.\*;      public class JavaIODemo  {  public static void main(String[] args)throws IOException  {  //Scanner scan = new Scanner(System.in);  DataInputStream dis = new DataInputStream(System.in);  String str;  System.out.print("\n\t Enter a string :");  str = dis.readLine();    int age;  System.out.println("\n\t Enter age :"); age = Integer.parseInt(dis.readLine());    System.out.println("\n\t Str = " + str);  System.out.println("\n\t Age = " + age);  }  } |

Program N0.

137

The **readLine()** method of **DataInputStream** reads a line at a time and returns as a string, irrespective of what the line contains. Depending on the input value, the string is to be parsed into an **int** or **double** etc.

## Keyboard input using InputStreamReader



By wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a BufferedReader, we can read input from the user in the command line. Here’s an example:

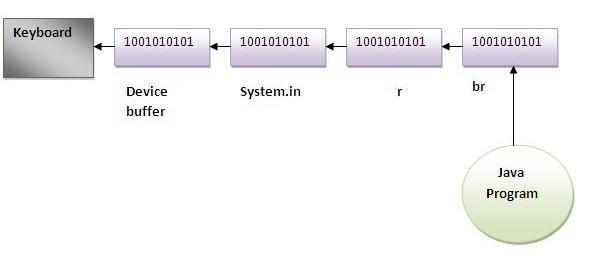
**InputStreamReader** class can be used to read data from keyboard.It performs two tasks:

1. connects to input stream of keyboard
2. converts the byte-oriented stream into character-oriented stream

**BufferedReader** class can be used to read data line by line by readLine() method.

InputStreamReader r = new InputStreamReader(System.in)

BufferedReader br = new BufferedReader(r)



* The **System.in** is a byte stream and cannot be chained to BufferedReader as BufferedReader is a character stream
* **Note :** (this problem, we did not face with DataInputStream as DataInputStream and System.in are both byte streams)
* The byte stream **System.in** should be converted (wrapped) into a character stream and then passed to BufferedReader constructor. This is done by **InputStreamReader**.

| **Keyboard Input using InputStreamReader** |
| --- |
| import java.io.\*;      public class JavaIODemo1\_138  {  public static void main(String[] args)throws IOException  {  InputStreamReader in = new InputStreamReader(System.in); BufferedReader br = new BufferedReader(in);    String str;  System.out.print("\n\t Enter a string :");  str = br.readLine();    int age;  System.out.println("\n\t Enter age :"); age = Integer.parseInt(br.readLine());    System.out.println("\n\t Str = " + str);  System.out.println("\n\t Age = " + age);  }  } |

Program N0.

138

* InputStreamReader, is neither an input stream nor a reader. It is not a carrier of data ▪ It is simply a wrapper around input stream to give a reader functionality.
* The InputStreamReader is used to link an input stream with character stream on reading-side

**Advantages:** The input is buffered for efficient reading.

**Drawbacks:** The wrapping code is hard to remember.

## Keyboard Input Using Scanner Class



The main purpose of the **Scanner** class (available since Java 1.5) is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line. Here’s an example:

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your nationality: ");

String nationality = scanner.nextLine();

System.out.print("Enter your age: "); int age = scanner.nextInt(); **Advantages:**

* Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
* Regular expressions can be used to find tokens.

**Drawbacks:**

• The reading methods are not synchronized.

## Keyboard input using Console class



The Java Console class is used to get input from console. It provides methods to read text and password. If you read password using Console class, it will not be displayed to the user. System class provides a static method console() that returns the unique instance of Console class.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

### Methods of Console class

| **Method** | **Description** |
| --- | --- |
| **public String readLine()** | It is used to read a single line of text from the console. |
| **public String readLine(String fmt,Object... args)** | It provides a formatted prompt then reads the single line of text from the console. |
| **public char[] readPassword()** | It is used to read password that is not being displayed on the console. |
| **public char[] readPassword(String fmt,Object... args)** | it provides a formatted prompt then reads the password that is not being displayed on the console |

**Advantages:**

* Reading password without echoing the entered characters.
* Reading methods are synchronized.
* Format string syntax can be used.

**Drawbacks:**

* Does not work in non-interactive environment (such as in an IDE).

| **Keyboard Input using Console class** |
| --- |
| import java.io.\*;    public class ReadStringTest  {  public static void main(String args[])  {  Console c=System.console();    System.out.print("Enter your name: ");  String n=c.readLine();    System.out.print("Enter password: "); char[] ch=c.readPassword();  String pass=String.valueOf(ch);//converting char array into string    System.out.println("Welcome "+n);  System.out.println("Password is: "+pass);  }  } |

Program N0.

138

java.io.PrintStream class:



The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException

### Methods of PrintStream class

* public void print(DataType value): it prints the specified value.
* public void println(DataType value): it prints the specified value and terminates the line

| **Write Ouput on Console Screen using PrintStream** |
| --- |
| import java.io.\*;    public class PrintStreamDemo  {  public static void main(String args[])  {  PrintStream pout = new PrintStream(System.out);  pout.println("Helloworld");  pout.print("This is PrintStream Example");    }  } |

Program N0.

138

## File and Directory



### Class java.io.File

The class java.io.File can represents either a **file** or a **directory**.

A **path string** is used to locate a **file** or a **directory**. Unfortunately, path strings are system dependent, e.g., "c:\myproject\java\Hello.java" in Windows or "/myproject/java/Hello.java" in Unix/Mac.

* Windows use back-slash '\' as the directory separator; while Unixes/Mac use forward-slash '/'.
* Windows use semi-colon ';' as path separator to separate a list of paths; while Unixes/Mac use colon ':'.
* Windows use "\r\n" as line delimiter for text file; while Unixes use "\n" and Mac uses "\r".
* The "c:\" or "\" is called the *root*. Windows supports multiple roots, each maps to a drive (e.g., "c:\", "d:\"). Unixes/Mac has a single root ("\")

A path could be *absolute* (beginning from the root) or *relative* (which is relative to a reference directory). Special notations "." and ".." denote the current directory and the parent directory, respectively.

The java.io.File class maintains these system-dependent properties, for you to write programs that are portable:

* **Directory Separator**: in static fields File.separator (as String) and File.separatorChar. [They failed to follow the Java naming convention for constants adopted since JDK 1.2.] As mentioned, Windows use backslash '\'; while Unixes/Mac use forward slash '/'.
* **Path Separator:** in static fields File.pathSeparator (as String) and File.pathSeparatorChar. As mentioned, Windows use semi-colon ';' to separate a list of paths; while Unixes/Mac use colon ':'.

You can construct a File instance with a path string or URI, as follows. Take note that the physical file/directory may or may not exist. A file URL takes the form of file://..., e.g., file:///d:/docs/programming/java/test.html.

public File(String pathString)

public File(String parent, String child) public File(File parent, String child)

**// Constructs a File instance based on the given path string.**

public File(URI uri)

**// Constructs a File instance by converting from the given file-URI "file://...."**

For examples,

File file = new File("in.txt"); **// A file relative to the current working directory**

File file = new File("d:\\myproject\\java\\Hello.java"); **// A file with absolute path** File dir = new File("c:\\temp"); **// A directory**

For applications that you intend to distribute as JAR files, you should use the URL class to reference the resources, as it can reference disk files as well as JAR'ed files , for example,

java.net.URL url = this.getClass().getResource("icon.png");

## Methods of java.io.File class



| **Method Name** | **Description** |
| --- | --- |
| public boolean **exists**() | Tests if this file/directory exists. |
| public long **length**() | Returns the length of this file. |
| public boolean **isDirectory**() | Tests if this instance is a directory. |
| public boolean **isFile**() | Tests if this instance is a file. |
| public boolean **canRead**() | Tests if this file is readable. |
| public boolean **canWrite**() | Tests if this file is writable. |
| public boolean **delete**() | Deletes this file/directory. |
| public void **deleteOnExit**() | Deletes this file/directory when the program terminates. |
| public boolean **renameTo**(File *dest*) | Renames this file. |
| public boolean **mkdir**() | Makes (Creates) this directory. |
| public Boolean **createNewFile**() | Crates new file |

## List Directory



For a directory, you can use the following methods to list its contents: public String[] **list**() // List the contents of this directory in a String-array public File[] **listFiles**() // List the contents of this directory in a File-array

### Example: The following program recursively lists the contents of a given directory

import java.io.File;

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

File dir = new File("d:\\myproject\\test"); // Escape sequence needed for '\' listRecursive(dir);

}

public static void listRecursive(File dir) { if (dir.isDirectory()) {

File[] items = dir.listFiles(); for (File item : items) {

System.out.println(item.getAbsoluteFile()); if (item.isDirectory())

listRecursive(item); // Recursive call

}

}

}

}

## List Directory with Filter



You can apply a filter to list() and listFiles(), to list only files that meet a certain criteria.

public String[] **list**(FilenameFilter *filter*) public File[] **listFiles**(FilenameFilter *filter*) public File[] **listFiles**(FileFilter *filter*)

**The interface java.io.FilenameFilter declares one abstract method:**

public boolean **accept**(File *dirName*, String *fileName*)

The list() and listFiles() methods does a *call-back* to accept() for each of the file/sub-directory produced. You can program your filtering criteria in accept(). Those files/sub-directories that result in a false return will be excluded.

Example: The following program lists only files that meet a certain filtering criteria.

**// List files that end with ".java"**

import java.io.File; import java.io.FilenameFilter; public class **ListDirectoryWithFilter** { public static void main(String[] args) {

File dir = new File("."); // current working directory if (dir.isDirectory()) {

// List only files that meet the filtering criteria

// programmed in accept() method of FilenameFilter.

String[] files = dir.list(**new FilenameFilter() { public boolean accept(File dir, String file) {**

**return file.endsWith(".java");**

**}**

**});** // an anonymous inner class as FilenameFilter for (String file : files) {

System.out.println(file);

}

}

}

}

## Character-Based I/O



| **Stream class** | **Description** |
| --- | --- |
| Reader | Abstract class that define character stream input |
| Writer | Abstract class that define character stream output |
| BufferedReader | Handles buffered input stream. |
| BufferedWriter | Handles buffered output stream. |
| FileReader | Input stream that reads from file. |
| FileWriter | Output stream that writes to file. |
| InputStreamReader | Input stream that translate byte to character |
| OutputStreamWriter | Output stream that translate character to byte. |
| PrintWriter | Output Stream that contain print() and println() method. |

## Abstract superclass Reader and Writer



The abstract superclass Reader operates on char. It declares an abstract method read() to read one character from the input source. read() returns the character as an int between 0 to 65535 (a char in Java can be treated as an unsigned 16-bit integer); or -1 if end-of-stream is detected; or throws an IOException if I/O error occurs. There are also two variations of read() to read a block of characters into char-array public abstract int read() throws IOException

public int read(char[] chars, int offset, int length) throws IOException public int read(char[] chars) throws IOException

**Writer** is the abstract class for writing character streams. It implements the following fundamental methods:

* write(int): writes a single character.
* write(char[]): writes an array of characters.
* write(String): writes a string. ▪ close(): closes the stream.

## OutputStreamWriter, FileWriter, BufferedWriter

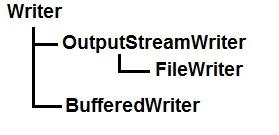


OutputStreamWriter is a bridge from byte streams to character streams. Characters are encoded into bytes using a specified charset. The charset can be default character encoding of the operating system, or can be specified explicitly when creating an OutputStreamWriter.

FileWriter is a convenient class for writing text files using the default character encoding of the operating system.

BufferedWriter writes text to a character stream with efficiency (characters, arrays and strings are buffered to avoid frequently writing to the underlying stream) and provides a convenient method for writing a line separator: newLine().

The following diagram show relationship of these writer classes in the java.io package:



### Character Encoding and Charset

When constructing a reader or writer object, the default character encoding of the operating system is used (e.g.

Cp1252 on Windows):

FileReader reader = new FileReader("MyFile.txt");

FileWriter writer = new FileWriter("YourFile.txt");

So if we want to use a specific charset, use an **InputStreamReader** or **OutputStreamWriter** instead. For example:

InputStreamReader reader = new InputStreamReader(

new FileInputStream("MyFile.txt"), "UTF-16");

That creates a new reader with the Unicode character encoding UTF-16.

And the following statement constructs a writer with the UTF-8 encoding:

OutputStreamWriter writer = new OutputStreamWriter(

new FileOutputStream("YourFile.txt"), "UTF-8");

In case we want to use a **BufferedReader**, just wrap the **InputStreamReader** inside, for example:

InputStreamReader reader = new InputStreamReader( new FileInputStream("MyFile.txt"), "UTF-16"); BufferedReader bufReader = new BufferedReader(reader);

And for a **BufferedWriter** example:

OutputStreamWriter writer = new OutputStreamWriter(

new FileOutputStream("YourFile.txt"), "UTF-8");

BufferedWriter bufWriter = new BufferedWriter(writer);

## Reading from Text File Example



The following small program reads every single character from the file MyFile.txt and prints all the characters to the output console:

import java.io.FileReader; import java.io.IOException;

public class TextFileReadingExample1 { public static void main(String[] args) { try {

FileReader reader = new FileReader("MyFile.txt"); int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

The following example reads a text file with assumption that the encoding is UTF-16:

import java.io.FileInputStream; import java.io.IOException; import java.io.InputStreamReader;

public class TextFileReadingExample2 { public static void main(String[] args) { try {

FileInputStream inputStream = new FileInputStream("MyFile.txt"); InputStreamReader reader = new InputStreamReader(inputStream, "UTF-16"); int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

## Buffered I/O Character-Streams - BufferedReader & BufferedWriter



BufferedReader and BufferedWriter can be stacked on top of FileReader/FileWriter or other character streams to perform buffered I/O, instead of character-by-character. BufferedReader provides a new method readLine(), which reads a line and returns a String

And the following example uses a **BufferedReader** to read a text file line by line (this is the most efficient and preferred way):

import java.io.BufferedReader; import java.io.FileReader; import java.io.IOException;

public class TextFileReadingExample3 { public static void main(String[] args) { try {

FileReader reader = new FileReader("MyFile.txt");

BufferedReader bufferedReader = new BufferedReader(reader);

String line;

while ((line = bufferedReader.readLine()) != null) {

System.out.println(line);

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

## Writing to Text File Example



In the following example, a **FileWriter** is used to write two words “Hello World” and “Good Bye!” to a file named MyFile.txt:

import java.io.FileWriter; import java.io.IOException;

public class TextFileWritingExample1 { public static void main(String[] args) { try {

FileWriter writer = new FileWriter("MyFile.txt", true); writer.write("Hello World");

writer.write("\r\n"); // write new line writer.write("Good Bye!"); writer.close(); } catch (IOException e) {

e.printStackTrace();

}

} }

Note that, a writer uses default character encoding of the operating system by default. It also creates a new file if not exits, or overwrites the existing one. If you want to append text to an existing file, pass a boolean flag of true to constructor of the writer class:

FileWriter writer = new FileWriter("MyFile.txt", true);

The following example uses a **BufferedReader** that wraps a **FileReader** to append text to an existing file:

import java.io.BufferedWriter;

import java.io.FileWriter; import java.io.IOException; public class TextFileWritingExample2 { public static void main(String[] args) { try {

FileWriter writer = new FileWriter("MyFile.txt", true);

BufferedWriter bufferedWriter = new BufferedWriter(writer);

bufferedWriter.write("Hello World"); bufferedWriter.newLine();

bufferedWriter.write("See You Again!");

bufferedWriter.close(); } catch (IOException e) {

e.printStackTrace();

}

} }

This is the preferred way to write to text file because the **BufferedReader** provides efficient way for writing character streams.

## InputStreamReader and OutputStreamWriter



As mentioned, Java internally stores characters (char type) in 16-bit UCS-2 character set. But the external data source/sink could store characters in other character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes. The FileReader/FileWriter introduced earlier uses the default charset for decoding/encoding, resulted in non-portable programs.

InputStreamReader and OutputStreamWriter are considered to be byte-to-character "bridge" streams.

You can choose the character set in the InputStreamReader's constructor: public **InputStreamReader**(InputStream *in*) // Use default charset public **InputStreamReader**(InputStream *in*, String *charsetName*) throws UnsupportedEncodingException public **InputStreamReader**(InputStream *in*, Charset *cs*)

You can list the available charsets via static method java.nio.charset.Charset.availableCharsets(). The commonly-used Charset names supported by Java are:

* "US-ASCII": 7-bit ASCII (aka ISO646-US)
* "ISO-8859-1": Latin-1
* "UTF-8": Most commonly-used encoding scheme for Unicode
* "UTF-16BE": Big-endian (big byte first) (big-endian is usually the default)
* "UTF-16LE": Little-endian (little byte first)
* "UTF-16": with a 2-byte BOM (Byte-Order-Mark) to specify the byte order. FE FF indicates big-endian, FF FE indicates little-endian.

As the InputStreamReader/OutputStreamWriter often needs to read/write in multiple bytes, it is best to wrap it with a BufferedReader/BufferedWriter.

And the following example specifies specific character encoding (UTF-16) when writing to the file:

import java.io.BufferedWriter; import java.io.FileOutputStream; import java.io.IOException; import java.io.OutputStreamWriter; public class TextFileWritingExample3 { public static void main(String[] args) { try {

FileOutputStream outputStream = new FileOutputStream

("MyFile.txt");

OutputStreamWriter outputStreamWriter = new OutputStreamWriter

(outputStream, "UTF-16");

BufferedWriter bufferedWriter = new BufferedWriter

(outputStreamWriter);

bufferedWriter.write("Hello world!"); bufferedWriter.newLine();

bufferedWriter.write("Welcome to my program");

bufferedWriter.close(); } catch (IOException e) {

e.printStackTrace();

}

}

}

This program writes some Unicode string to the specified text file.

**NOTE:** From Java 7, you can use try-with-resources statement to simplify the code of opening and closing the reader/writer. For example:

try (FileReader reader = new FileReader("MyFile.txt")) { int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

}

} catch (IOException e) {

e.printStackTrace();

}

## java.io.PrintStream & java.io.PrintWriter



The byte-based java.io.printSteam supports convenient printing methods such as print() and println() for printing primitives and text string. Primitives are converted to their string representation for printing. The printf() and format() were introduced in JDK 1.5 for formatting output with former specifiers. printf() and format() are identical.

A PrintStream never throws an IOException. Instead, it sets an internal flag which can be checked via the checkError() method. A PrintStream can also be created to flush the output automatically. That is,

the flush()method is automatically invoked after a byte array is written, one of the println() methods is invoked, or after a newline ('\n') is written.

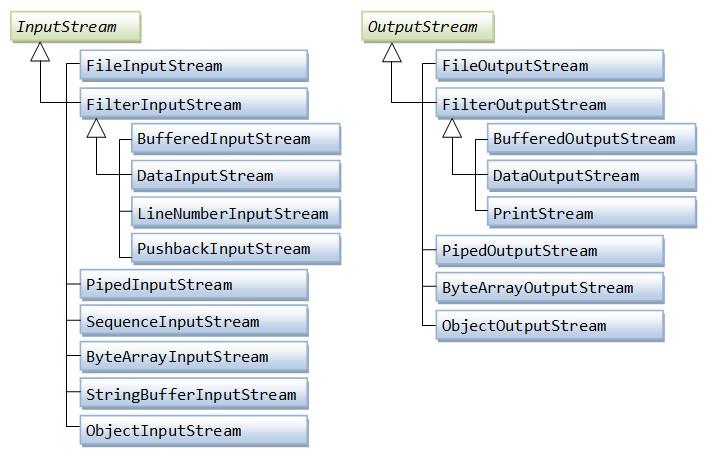
The standard output and error streams (System.out and System.err) belong to PrintStream.

All characters printed by a PrintStream are converted into bytes using the default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes.

The character-stream PrintWriter is similar to PrintStream, except that it write in characters instead of bytes. The PrintWriter also supports all the convenient printing methods print(), println(), printf() and format(). It never throws an IOException and can optionally be created to support automatic flushing

## InputStream and OutputStream





Some important Byte stream classes.

| **Stream class** | **Description** |
| --- | --- |
| InputStream | Abstract class that describe stream input. |
| OutputStream | Abstract class that describe stream output. |
| PrintStream | Output Stream that contain print() and println() method |
| BufferedInputStream | Used for Buffered Input Stream. |
| BufferedOutputStream | Used for Buffered Output Stream. |
| DataInputStream | Contains method for reading java standard datatype |
| DataOutputStream | An output stream that contain method for writing java standard data type |
| FileInputStream | Input stream that reads from a file |
| FileOutputStream | Output stream that write to a file. |



## InputStream class



InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Commonly used methods of InputStream class

| **Method** | **Description** |
| --- | --- |
| public abstract int read()throws IOException: | Reads the next byte of data from the input stream. It returns -1 at the end of file. |
| public int available()throws IOException: | Returns an estimate of the number of bytes that can be read from the current input stream. |
| public void close()throws IOException: | It is used to close the current input stream. |

## OutputStream class



OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

### Commonly used methods of OutputStream class

| **Method** | **Description** |
| --- | --- |
| public void write(int)throws IOException: | It is used to write a byte to the current output stream. |
| public void write(byte[])throws IOException: | It is used to write an array of byte to the current output stream. |
| public void flush()throws IOException: | It flushes the current output stream. |
| public void close()throws IOException: | It is used to close the current output stream. |

### Reading from an InputStream

The abstract superclass InputStream declares an abstract method read() to read one data-byte from the input source:

public abstract int **read**() throws IOException **The read() method:**

▪ returns the input byte read as an int in the range of 0 to 255, or ▪ returns -1 if "end of stream" condition is detected, or ▪ throws an IOException if it encounters an I/O error.

The read() method returns an int instead of a byte, because it uses -1 to indicate end-of-stream.

The read() method blocks until a byte is available, an I/O error occurs, or the "end-of-stream" is detected. The term "block" means that the method (and the program) will be suspended. The program will resume only when the method returns.

Two variations of read() methods are implemented in the InputStream for reading a block of bytes into a byte-array. It returns the number of bytes read, or -1 if "end-of-stream" encounters.

public int **read**(byte[] *bytes*, int *offset*, int *length*) throws IOException

// Read "length" number of bytes, store in bytes array starting from offset of index.

public int **read**(byte[] *bytes*) throws IOException

// Same as read(bytes, 0, bytes.length)

### Writing to an OutputStream

Similar to the input counterpart, the abstract superclass OutputStream declares an abstract method write() to write a data-byte to the output sink. write() takes an int. The least-significant byte of the int argument is written out; the upper 3 bytes are discarded. It throws an IOException if I/O error occurs (e.g., output stream has been closed).

public void abstract void **write**(int *unsignedByte*) throws IOException

Similar to the read(), two variations of the write() method to write a block of bytes from a byte-array are implemented:

public void **write**(byte[] *bytes*, int *offset*, int *length*) throws IOException // Write "*length*" number of bytes, from the *bytes* array starting from offset of *index*.

public void **write**(byte[] *bytes*) throws IOException

// Same as write(*bytes*, 0, *bytes*.length)

### Opening & Closing I/O Streams

You open an I/O stream by constructing an instance of the stream. Both the InputStream and the OutputStream provides a close() method to close the stream, which performs the necessary clean-up operations to free up the system resources.

public void **close**() throws IOException // close this Stream

It is a good practice to explicitly close the I/O stream, by running close() in the finally clause of try-catchfinally to free up the system resources immediately when the stream is no longer needed. This could prevent serious resource leaks. Unfortunately, the close() method also throws aIOException, and needs to be enclosed in a nested try-catch statement, as follows. This makes the codes somehow ugly.

FileInputStream in = null;

...... try {

in = new FileInputStream(...); // Open stream

......

......

} catch (IOException ex) { ex.printStackTrace();

} finally { // always close the I/O streams try {

if (in != null) **in.close()**; } catch (IOException ex) { ex.printStackTrace();

}

}

## FileInputStream & FileOutputStream



FileInputStream and FileOutputStream are concrete implementations to the abstract classes InputStream and OutputStream, to support I/O from disk files.

### FileOutputStream class

Java FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream.Instead, for character-oriented data, prefer FileWriter.But you can write byte-oriented as well as character-oriented dat

| **File OutputStream Demo** |
| --- |
| import java.io.\*;  public class FileOutputStreamDemo  {  public static void main(String args[])  { try  {  FileOutputStream fout=new FileOutputStream("abc.txt"); String s="Sachin Tendulkar is my favourite player"; byte b[]=s.getBytes();//converting string into byte array fout.write(b); fout.close();  System.out.println("success...");  }  catch(Exception e)  {  System.out.println(e);  }  }  } |

Program N0.

### FileInputStream class

Java FileInputStream class obtains input bytes from a file.It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data for example to read image, audio, video etc

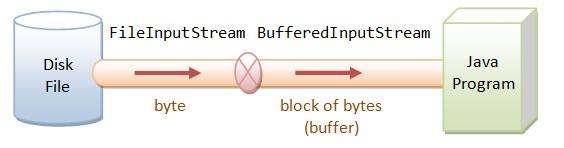
| **File InputStream Demo** |
| --- |
| import java.io.\*; public class SimpleRead  {  public static void main(String args[])  {  try  {  FileInputStream fin=new FileInputStream("abc.txt"); int i=0;  while((i=fin.read())!=-1)  {  System.out.println((char)i);  }  fin.close();  }  catch(IOException e)  {  System.out.println(e);  }  }  } |

Program N0.

## BufferedInputStream & BufferedOutputStream



The I/O streams are often layered or chained with other I/O streams, for purposes such as buffering, filtering, or dataformat conversion (between raw bytes and primitive types). For example, we can layer a BufferedInputStream to a FileInputStream for buffered input,



The read()/write() method in InputStream/OutputStream are designed to read/write a single byte of data on each call. This is grossly inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations). Buffering, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

FileInputStream/FileOutputStream is not buffered. It is often chained to

a BufferedInputStream or BufferedOutputStream, which provides the buffering. To chain the streams together, simply pass an instance of one stream into the constructor of another stream. For example, the following codes chain a FileInputStream to a BufferedInputStream, and finally, a DataInputStream:

FileInputStream fileIn = new FileInputStream("in.dat");

BufferedInputStream bufferIn = new BufferedInputStream(fileIn);

DataInputStream dataIn = new DataInputStream(bufferIn);

// or

DataInputStream in = new DataInputStream( new BufferedInputStream( new FileInputStream("in.dat")));

| **Copying a file byte-by-byte without Buffering.** |
| --- |
| import java.io.\*;  public class **FileCopyNoBuffer** { // Pre-JDK 7 public static void main(String[] args) {  String inFileStr = "test-in.jpg";  String outFileStr = "test-out.jpg";  FileInputStream in = null; FileOutputStream out = null; long startTime, elapsedTime; // for speed benchmarking  // Print file length  **File fileIn = new File(inFileStr);**  System.out.println("File size is " + **fileIn.length()** + " bytes");  try { **in = new FileInputStream(inFileStr)**; |

Program N0.

| **out = new FileOutputStream(outFileStr);**  startTime = System.nanoTime(); int byteRead;  // Read a raw byte, returns an int of 0 to 255.  while ((**byteRead = in.read()**) != -1) {  // Write the least-significant byte of int, drop the upper 3 bytes **out.write(byteRead);**  }  elapsedTime = System.nanoTime() - startTime;  System.out.println("Elapsed Time is" + (elapsedTime/1000000.0)+  " msec");  } catch (IOException ex) { ex.printStackTrace();  } finally { // always close the I/O streams try { if (in != null) **in.close()**; if (out != null) **out.close()**; } catch (IOException ex) { ex.printStackTrace();  }  }  }  } |
| --- |

This example copies a file by reading a byte from the input file and writing it to the output file. It uses FileInputStream and FileOutputStream directly without buffering. Notice that most the I/O methods "throws" IOException, which must be caught or declared to be thrown. The methodclose() is programmed inside the finally clause. It is guaranteed to be run after try or catch. However, method close() also throws an IOException, and therefore must be enclosed inside a nested try-catch block, which makes the codes a little ugly.

We used System.nanoTime(), which was introduced in JDK 1.5, for a more accurate measure of the elapsed time, instead of the legacy not-so-precise System.currentTimeMillis(). The output shows that it took about 4 seconds to copy a 400KB file.

| **Copying a file with a Programmer-Managed Buffer.** |
| --- |
| import java.io.\*;  public class **FileCopyUserBuffer** { // Pre-JDK 7 public static void main(String[] args) { String inFileStr = "test-in.jpg";  String outFileStr = "test-out.jpg";  FileInputStream in = null; FileOutputStream out = null;  long startTime, elapsedTime; // for speed benchmarking  // Check file length  File fileIn = new File(inFileStr);  System.out.println("File size is " + fileIn.length() + " bytes");  try {  in = new FileInputStream(inFileStr); out = new FileOutputStream(outFileStr); startTime = System.nanoTime();  **byte[] byteBuf = new byte[4096];** // 4K byte-buffer int numBytesRead;  while ((numBytesRead = **in.read(byteBuf)**) != -1) { |

Program N0. 138

| **out.write(byteBuf, 0, numBytesRead)**;  }  elapsedTime = System.nanoTime() - startTime;  System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");  } catch (IOException ex) { ex.printStackTrace();  } finally { // always close the streams try {  if (in != null) in.close(); if (out != null) out.close();  } catch (IOException ex) { ex.printStackTrace(); }  }  }  } |
| --- |

This example again uses FileInputStream and FileOutputStream directly. However, instead of reading/writing one byte at a time, it reads/writes a 4KB block. This program took only 3 millisecond - a more than 1000 times speed-up compared with the previous example.

Larger buffer size, up to a certain limit, generally improves the I/O performance. However, there is a trade-off between speed-up the the memory usage. For file copying, a large buffer is certainly recommended. But for reading just a few bytes from a file, large buffer simply wastes the memory.

**re-write the program using JDK 1.7, and try on various buffer size on a much bigger file of 26MB.**

import java.io.\*;

public class **FileCopyUserBufferLoopJDK7** { public static void main(String[] args) { String inFileStr = "test-in.jpg"; String outFileStr = "test-out.jpg"; long startTime, elapsedTime; // for speed benchmarking

// Check file length

File fileIn = new File(inFileStr);

System.out.println("File size is " + fileIn.length() + " bytes");

int[] bufSizeKB = {1, 2, 4, 8, 16, 32, 64, 256, 1024}; // in KB int bufSize; // in bytes

for (int run = 0; run < bufSizeKB.length; ++run) { bufSize = bufSizeKB[run] \* 1024; try (FileInputStream in = new FileInputStream(inFileStr); FileOutputStream out = new FileOutputStream(outFileStr)) { startTime = System.nanoTime(); byte[] byteBuf = new byte[bufSize]; int numBytesRead; while ((numBytesRead = in.read(byteBuf)) != -1) { out.write(byteBuf, 0, numBytesRead);

} elapsedTime = System.nanoTime() - startTime;

System.out.printf("%4dKB: %6.2fmsec%n", bufSizeKB[run], (elapsedTime / 1000000.0));

//System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");

} catch (IOException ex) { ex.printStackTrace();

}

}

}

}

### Copying File with Buffered Streams

| **Copying a file with Buffered Streams.** |
| --- |
| import java.io.\*;  public class **FileCopyBufferedStream** { // Pre-JDK 7  public static void main(String[] args) { String inFileStr = "test-in.jpg";  String outFileStr = "test-out.jpg";  **BufferedInputStream in** = null; **BufferedOutputStream out** = null;  long startTime, elapsedTime; // for speed benchmarking  // Check file length  File fileIn = new File(inFileStr);  System.out.println("File size is " + fileIn.length() + " bytes");  try {  **in = new BufferedInputStream(new FileInputStream(inFileStr)); out = new BufferedOutputStream(new FileOutputStream(outFileStr));** startTime = System.nanoTime(); int byteRead;  while ((byteRead = **in.read()**) != -1) { // Read byte-by-byte from buffer  **out.write(byteRead)**;  }  elapsedTime = System.nanoTime() - startTime;  System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");  } catch (IOException ex) { ex.printStackTrace();  } finally { // always close the streams try {  if (in != null) in.close(); if (out != null) out.close();  } catch (IOException ex) { ex.printStackTrace(); }  }  }  } |

Program N0.

In this example, the **FileInputStream** with **BufferedInputStream**, **FileOutputStream** with **BufferedOutputStream**, and read/write byte-by-byte. The JRE decides on the buffer size. The program took 62 milliseconds, about 60 times speed-up compared with example 1, but slower than the programmer-managed buffer.

The JDK 1.7 version of the above example is as follows: import java.io.\*;

public class FileCopyBufferedStreamJDK7 { public static void main(String[] args) { String inFileStr = "test-in.jpg"; String outFileStr = "test-out.jpg";

long startTime, elapsedTime; // for speed benchmarking

// Check file length

File fileIn = new File(inFileStr);

System.out.println("File size is " + fileIn.length() + " bytes");

**try (BufferedInputStream in = new BufferedInputStream(new FileInputStream(inFileStr));**

**BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(outFileStr)))**

{ startTime = System.nanoTime(); int byteRead; while ((byteRead = in.read()) != -1) { out.write(byteRead);

} elapsedTime = System.nanoTime() - startTime;

System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");

} catch (IOException ex) { ex.printStackTrace();

}

}

}

Formatted Data-Streams:

## DataInputStream & DataOutputStream



The DataInputStream and DataOutputStream can be stacked on top of any InputStream and OutputStream to parse the raw bytes to perform I/O operations in the desired data format, such as int and double.

To use DataInputStream for formatted input, you can chain up the input streams as follows:

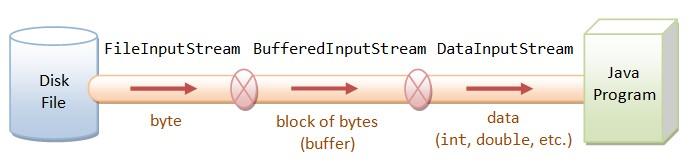
DataInputStream in = new DataInputStream( new BufferedInputStream( new FileInputStream("in.dat")));

DataInputStream implements DataInput interface, which provides methods to read formatted primitive data and String, such as:

| **Method** | | **Description** | |
| --- | --- | --- | --- |
| public final int **readInt**() throws IOExcpetion; | | Read 4 bytes and convert into int | |
| public final double **readDoube**() throws IOExcpetion; | | Read 8 bytes and convert into double | |
| public final byte **readByte**() throws IOExcpetion; | |  | |
| public final char **readChar**() throws IOExcpetion; | |  | |
| public final short **readShort**() throws IOExcpetion; | |  | |
| public final long **readLong**() throws IOExcpetion; | |  | |
| public final boolean **readBoolean**() throws IOExcpetion; | | Read 1 byte. Convert to false if zero | |
| public final float **readFloat**() throws IOExcpetion; | |  | |
|  | |  | |
| public final int **readUnsignedByte**() throws IOExcpetion; | | Read 1 byte in [0, 255] upcast to int | |
| public final int **readUnsignedShort**() throws IOExcpetion; | | Read 2 bytes in [0, 65535], same as char, upcast to int | |
|  | public final void **readFully**(byte[] *b*, int *off*, int *len*) throws IOException; | |  | |
|  | public final void **readFully**(byte[] *b*) throws IOException; | |  | |
|  |  | |  | |
|  | public final String **readLine**() throws IOException; | | Read a line (until newline), convert each byte into a char - no unicode support | |
|  | public final String **readUTF**() throws IOException; | | read a UTF-encoded string with first two bytes indicating its UTF bytes length | |
|  |  | |  | |
|  | public final int **skipBytes**(int n) | | Skip a number of bytes | |

Similarly, you can stack the DataOutputStream as follows:

DataOutputStream out = new DataOutputStream( new BufferedOutputStream( new FileOutputStream("out.dat")));

 DataOutputStream implements DataOutput interface, which provides methods to write formatted primitive data and String. For examples,

| **Method** | **Description** |
| --- | --- |
| public final void **writeInt**(int *i*) throws IOExcpetion; | Write the int as 4 bytes |
| public final void **writeFloat**(float *f*) throws IOExcpetion; |  |
| public final void **writeDoube**(double *d*) throws IOExcpetion; | Write the double as 8 bytes |
| public final void **writeByte**(int *b*) throws IOExcpetion; | least-significant byte |
| public final void **writeShort**(int *s*) throws IOExcpetion; | two lower bytes |
| public final void **writeLong**(long *l*) throws IOExcpetion; |  |
| public final void **writeBoolean**(boolean *b*) throws IOExcpetion; |  |
| public final void **writeChar**(int *i*) throws IOExcpetion; |  |
|  |  |
| public final void **writeByte**s(String *str*) throws IOExcpetion; | least-significant byte of each char |
| public final void **writeChars**(String *str*) throws IOExcpetion; | Write String as UCS-2 16-bit char, Big-endian (big byte first) |
| public final void **writeUTF**(String *str*) throws IOException; | Write String as UTF, with first two bytes indicating UTF bytes length |
| public final void **write**(byte[] *b*, int *off*, int *len*) throws IOException  public final void **write**(byte[] *b*) throws IOException |  |
| public final void **write**(int *b*) throws IOException | Write the least-significant byte |

Example: The following program writes some primitives to a disk file. It then reads the raw bytes to check how the primitives were stored. Finally, it reads the data as primitives.

import java.io.\*;

public class TestDataIOStream {

public static void main(String[] args) {

String filename = "data-out.dat";

String message = "Hi,您好!";

// Write primitives to an output file

try (DataOutputStream out = new DataOutputStream( new BufferedOutputStream(

new FileOutputStream(filename)))) { out.writeByte(127); out.writeShort(0xFFFF); // -1 out.writeInt(0xABCD); out.writeLong(0x1234\_5678); // JDK 7 syntax out.writeFloat(11.22f); out.writeDouble(55.66); out.writeBoolean(true); out.writeBoolean(false);

for (int i = 0; i < message.length(); ++i) { out.writeChar(message.charAt(i));

}

out.writeChars(message); out.writeBytes(message); out.flush();

} catch (IOException ex) { ex.printStackTrace();

}

// Read raw bytes and print in Hex try (BufferedInputStream in = new BufferedInputStream( new FileInputStream(filename))) { int inByte;

while ((inByte = in.read()) != -1) {

System.out.printf("%02X ", inByte); // Print Hex codes }

System.out.printf("%n%n"); } catch (IOException ex) { ex.printStackTrace();

}

// Read primitives try (DataInputStream in = new DataInputStream( new BufferedInputStream(

new FileInputStream(filename)))) { System.out.println("byte: " + in.readByte());

System.out.println("short: " + in.readShort());

System.out.println("int: " + in.readInt());

System.out.println("long: " + in.readLong());

System.out.println("float: " + in.readFloat());

System.out.println("double: " + in.readDouble());

System.out.println("boolean: " + in.readBoolean()); System.out.println("boolean: " + in.readBoolean()); System.out.print("char: ");

for (int i = 0; i < message.length(); ++i) {

System.out.print(in.readChar());

}

System.out.println();

System.out.print("chars: ");

for (int i = 0; i < message.length(); ++i) {

System.out.print(in.readChar());

}

System.out.println();

System.out.print("bytes: ");

for (int i = 0; i < message.length(); ++i) {

System.out.print((char)in.readByte());

}

System.out.println(); } catch (IOException ex) { ex.printStackTrace();

}

}

}

Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the [DataInput](https://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html) interface or the [DataOutput](https://docs.oracle.com/javase/8/docs/api/java/io/DataOutput.html) interface. This section focuses on the most widely-used implementations of these interfaces, [DataInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html) and

[DataOutputStream.](https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html)

The [DataStreams](https://docs.oracle.com/javase/tutorial/essential/io/examples/DataStreams.java) example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:

| **Order in record** | **Data type** | **Data description** | **Output Method** | **Input Method** | **Sample Value** |
| --- | --- | --- | --- | --- | --- |
| 1 | Double | Item price | DataOutputStream.writeDouble | DataInputStream.readDouble | 19.99 |
| 2 | Int | Unit count | DataOutputStream.writeInt | DataInputStream.readInt | 12 |
| 3 | String | Item description | DataOutputStream.writeUTF | DataInputStream.readUTF | "Java T-  Shirt" |

Let's examine crucial code in DataStreams. First, the program defines some constants containing the name of the data file and the data that will be written to it:

static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };

static final int[] units = { 12, 8, 13, 29, 50 }; static final String[] descs = {

"Java T-shirt",

"Java Mug",

"Duke Juggling Dolls",

"Java Pin",

"Java Key Chain" };

Then DataStreams opens an output stream. Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, DataStreams provides a buffered file output byte stream.

out = new DataOutputStream(new BufferedOutputStream( new FileOutputStream(dataFile)));

DataStreams writes out the records and closes the output stream.

for (int i = 0; i < prices.length; i ++) { out.writeDouble(prices[i]); out.writeInt(units[i]); out.writeUTF(descs[i]);

}

The writeUTF method writes out String values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now DataStreams reads the data back in again. First it must provide an input stream, and variables to hold the input data. Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

in = new DataInputStream(new

BufferedInputStream(new FileInputStream(dataFile))); double price; int unit; String desc; double total = 0.0;

Now DataStreams can read each record in the stream, reporting on the data it encounters.

try {

while (true) {

price = in.readDouble(); unit = in.readInt(); desc = in.readUTF();

System.out.format("You ordered %d" + " units of %s at $%.2f%n", unit, desc, price); total += unit \* price;

}

} catch (EOFException e) { }

Notice that DataStreams detects an end-of-file condition by catching [EOFException,](https://docs.oracle.com/javase/8/docs/api/java/io/EOFException.html) instead of testing for an invalid return value. All implementations of DataInput methods use EOFException instead of return values.

Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.

## Object Streams and Object Serialization



Data streams (DataInputStream and DataOutputStream) allow you to read and write primitive data (such as int, double) and String, rather than individual bytes. Just as data streams support I/O of primitive data types, object streams support I/O of objects

The object stream classes are [ObjectInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html) and [ObjectOutputStream.](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html) These classes implement [ObjectInput](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInput.html) and [ObjectOutput,](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutput.html) which are subinterfaces of DataInput and DataOutput.

Most of the core Java classes implement Serializable, such as all the wrapper classes, collection classes, and GUI classes. In fact, the only core Java classes that do not implement Serializable are ones that should not be serialized. Arrays of primitives or serializable objects are themselves serializable. transient & static

* static fields are not serialized, as it belongs to the class instead of the particular instance to be serialized.
* To prevent certain fields from being serialized, mark them using the keyword **transient**. This could cut down the amount of data traffic.
* The writeObject() method writes out the class of the object, the class signature, and values of non-

static and non-transient fields

## Object streams



(ObjectInputStream and ObjectOutputStream) go one step further to allow you to read and write entire objects (such as Date, ArrayList or any custom objects).

Object serialization is the process of representing a "particular state of an object" in a serialized bit-stream, so that the bit stream can be written out to an external device (such as a disk file or network). The bit-stream can later be reconstructed to recover the state of that object. Object serialization is necessary to save a state of an object into a disk file for persistence or sent the object across the network for applications such as Web Services, Distributed-object applications, and Remote Method Invocation (RMI).

In Java, object that requires to be serialized must

implement java.io.Serializable or java.io.Externalizable interface. Serializable interface is

an empty interface (or *tagged* interface) with nothing declared. Its purpose is simply to declare that particular object is serializable.

## ObjectInputStream & ObjectOutputStream



We use object streams to read and write Java objects in binary format. ObjectInputStreamand ObjectOutputStream are the main object stream classes provided by the Java File I/O API.

The ObjectOutputStream class implements the ObjectOutput interface that defines a method for writing an object to an output stream:

* writeObject(Object): writes an object to the underlying storage or stream. This method throws IOException if an I/O error occurs.

The process of writing an object to an output stream is called **Serialization**

The ObjectOutput interface extends from the DataOutput interface, which means an ObjectOutputStream inherits all behaviors of writing primitive types and Strings like a DataOutputStream.

Likewise, the ObjectInputStream class implements the ObjectInput interface that defines a method for reading an object from an input stream:

* **readObject():** reads and returns an object. This method throws ClassNotFoundException if the class of the serialized object cannot be found, and throws IOException if an I/O error occurs.

The process of reconstructing an object from an input stream is called **deserialization**.

The ObjectInput interface extends from the DataInput interface, which means an ObjectInputStream also has behaviors of reading primitive types and Strings like a DataInputStream.

Which kinds of object are eligible for serialization?

Note that only objects of classes that implement the [java.io.](http://java.io/)**Serializable** interface can be written to and read from an output/input stream. Serializable is a marker interface, which doesn’t define any methods. Only objects that are marked ‘serializable’ can be used with ObjectOutputStream and ObjectInputStream.

Most classes in Java (including Date and primitive wrappers Integer, Double, Long, etc) implement the Serializable interface. We have to implement this interface for our custom classes only, such as the Student class we see in the previous email about data streams.

If we attempt to write an object of a non-serializable class, we will get a java.io.NotSerializableException.

import java.util.Date; import java.io.Serializable;

public class Student implements Serializable { private String name; private Date birthday;

private boolean gender; // true is male, false is female private int age; private float grade; public Student() {

}

public Student(String name, Date birthday, boolean gender, int age, float grade) {  [this.name](http://this.name/) = name; this.birthday = birthday; this.gender = gender; this.age = age; this.grade = grade;

}

// getters and setters go here... }

As you can see, we make this class implements the Serializable interface and add another member variable call birthday which is of type java.util.Date.

Rewrite the StudentRecordWriter program to use an ObjectOutputStream like this:

import java.util.\*; import java.text.\*; import java.io.\*;

public class StudentRecordWriter { public static void main(String[] args) { if (args.length < 1) {

System.out.println("Please provide output file");

System.exit(0);

}

String outputFile = args[0];

DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy"); try (

ObjectOutputStream objectOutput

= new ObjectOutputStream(new FileOutputStream(outputFile)); ) {

List listStudent = new ArrayList<>();

listStudent.add(new Student("Alice", dateFormat.parse("02-15-1993"), false, 23, 80.5f));

listStudent.add(new Student("Brian", dateFormat.parse("10-03-1994"), true, 22, 95.0f));

listStudent.add(new Student("Carol", dateFormat.parse("08-22-1995"), false, 21, 79.8f)); for (Student student : listStudent) { objectOutput.writeObject(student);

}

} catch (IOException | ParseException ex) { ex.printStackTrace();

}

}

}

Run this program via command line:

java StudentRecordWriter Student.db

And rewrite the StudentRecordReader program like this:

import java.text.\*; import java.io.\*;

public class StudentRecordReader {

public static void main(String[] args) { if (args.length < 1) {

System.out.println("Please provide input file");

System.exit(0);

}

String inputFile = args[0];

DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy"); try (

ObjectInputStream objectInput

= new ObjectInputStream(new FileInputStream(inputFile));

){

while (true) {

Student student = (Student) objectInput.readObject();

System.out.print(student.getName() + "\t");

System.out.print(dateFormat.format(student.getBirthday()) + "\t");

System.out.print(student.getGender() + "\t");

System.out.print(student.getAge() + "\t");

System.out.println(student.getGrade());

}

} catch (EOFException eof) {

System.out.println("Reached end of file"); } catch (IOException | ClassNotFoundException ex) { ex.printStackTrace();

}

}

}

Run this program via command line:

java StudentRecordReader Student.db

Output:

Alice 02-15-1993 false 23 80.5

Brian 10-03-1994 true 22 95.0 Carol 08-22-1995 false 21 79.8

Reached end of file

## java.io.Externalizable Interface



The Serializable has a sub-interface called Externalizable, which you could used if you want to customize the way a class is serialized. Since Externalizable extends Serializable, it is also a Serializable and you could invoke readObject() and writeObject().

Externalizable declares two abstract methods:

void **writeExternal**(ObjectOutput *out*) throws IOException

void **readExternal**(ObjectInput *in*) throws IOException, ClassNotFoundException

ObjectOutput and ObjectInput are interfaces that are implemented by ObjectOutputStream and ObjectInputStream, which define

the writeObject() and readObject() methods, respectively. When an instance of Externalizable is passed to an ObjectOutputStream, the default serialization procedure is bypassed; instead, the stream calls the instance's writeExternal() method. Similarly, when an ObjectInputStream reads a Exteranlizabled instance, it uses readExternal() to reconstruct the instance.

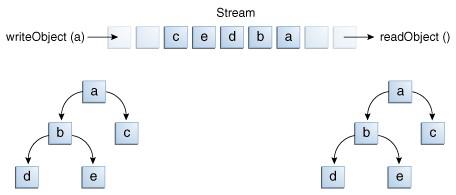
Externalizable is useful if you want complete control on how your objects shall be serialized/deserialized. For example, you could encrypt sensitive data before the object is serialized

## Output and Input of Complex Objects



The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking writeobject(a) writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

Object ob = new Object(); out.writeObject(ob); out.writeObject(ob);

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

Object ob1 = in.readObject();

Object ob2 = in.readObject();

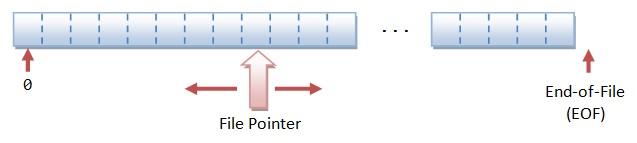
This results in two variables, ob1 and ob2, that are references to a single object.

## Random Access Files



All the I/O streams covered so far are one-way streams. That is, they are either read-only input stream or write-only output stream. Furthermore, they are all sequential-access (or serial) streams, meant for reading and writing data sequentially. Nonetheless, it is sometimes necessary to read a file record directly as well as modifying existing records or inserting new records. The class RandomAccessFile provides supports for non-sequential, direct (or random) access to a disk file. RandomAccessFile is a two-way stream, supporting both input and output operations in the same stream.

RandomAccessFile can be treated as a huge byte array. You can use a file pointer (of type long), similar to array index, to access individual byte or group of bytes in primitive types (such as int and double). The file pointer is located at 0 when the file is opened. It advances automatically for every read and write operation by the number of bytes processed.



In constructing a RandomAccessFile, you can use flags 'r' or 'rw' to indicate whether the file is "read-only" or "readwrite" access, e.g.,

RandomAccessFile f1 = new **RandomAccessFile**("filename", "r");

RandomAccessFile f2 = new **RandomAccessFile**("filename", "rw");

The following methods are available:

| **Method** | **Description** |
| --- | --- |
| public void seek(long pos) throws IOException | Positions the file pointer for subsequent read/write operation |
| public int skipBytes(int numBytes) throws IOException | Moves the file pointer forward by the specified number of bytes |
| public long getFilePointer() throws IOException | Gets the position of the current file pointer, in bytes, from the beginning of the file. |
| public long length() throws IOException | Returns the length of this file |

RandomAccessFile does not inherit from InputStream or OutputStream. However, it

implements DataInput and DataOutput interfaces (similar to DataInputStream andDataOutputStream). Therefore, you can use various methods to read/write primitive types to the file, e.g.,

public int **readInt**() throws IOException; public double **readDouble**() throws IOException; public void **writeInt**(int i) throws IOException; public void **writeDouble**(double d) throws IOException;

**Example:** Read and write records from a RandomAccessFile. (A student file consists of student record of name (String) and id (int)).

## Formatted-Text Input via java.util.Scanner



JDK 1.5 introduces java.util.Scanner class, which greatly simplifies formatted text input from input source (e.g., files, keyboard, network). Scanner, as the name implied, is a simple text scanner which can parse the input text into primitive types and strings using regular expressions. It first breaks the text input into tokens using a delimiter pattern, which is by default the white spaces (blank, tab and newline).

The tokens may then be converted into primitive values of different types using the various nextXxx() methods

(nextInt(), nextByte(), nextShort(), nextLong(), nextFloat(), nextDouble(), nextBoolean(), next() for String, and nextLine() for an input line). You can also use the hasNextXxx() methods to check for the availability of a desired input.

The commonly-used constructors are as follows. You can construct a Scanner to parse a bytebased InputStream (e.g., System.in), a disk file, or a given String.

// Scanner piped from a disk File

public Scanner(File source) throws FileNotFoundException

public Scanner(File source, String charsetName) throws FileNotFoundException

// Scanner piped from a byte-based InputStream, e.g., System.in

public Scanner(InputStream source)

public Scanner(InputStream source, String charsetName)

// Scanner piped from the given source string (NOT filename string)

public Scanner(String source)

**For examples,**

// Construct a Scanner to parse an int from keyboard

Scanner in1 = new Scanner(System.in); int i = in1.nextInt();

// Construct a Scanner to parse all doubles from a disk file

Scanner in2 = new Scanner(new File("in.txt")); // need to handle FileNotFoundException while (in2.hasNextDouble()) { double d = in.nextDouble();

}

// Construct a Scanner to parse a given text string

Scanner in3 = new Scanner("This is the input text String"); while (in3.hasNext()) {

String s = in.next();

}

**Example 1:** The most common usage of Scanner is to read primitive types and String form the keyboard (System.in), as follows:

import java.util.Scanner; public class **TestScannerSystemIn** { public static void main(String[] args) {

**Scanner in = new Scanner(System.in);**

System.out.print("Enter an integer: "); int anInt = **in.nextInt()**;

System.out.println("You entered " + anInt);

System.out.print("Enter a floating-point number: "); double aDouble = **in.nextDouble()**;

System.out.println("You entered " + aDouble);

System.out.print("Enter 2 words: ");

String word1 = **in.next()**; // read a string delimited by white space String word2 = **in.next()**; // read a string delimited by white space

System.out.println("You entered " + word1 + " " + word2);

**in.nextLine()**; // flush the "enter" before the next readLine()

System.out.print("Enter a line: ");

String line = **in.nextLine()**; // read a string up to line delimiter

System.out.println("You entered " + line);

} }

The nextXxx() methods throw InputMismatchException if the next token does not match the type to be parsed.

**Example 2:** You can easily modify the above program to read the inputs from a text file, instead of keyboard (System.in).

import java.util.Scanner; import java.io.\*; public class **TestScannerFile** {

public static void main(String[] args) throws FileNotFoundException { Scanner in = new Scanner(**new File("in.txt")**);

System.out.print("Enter an integer: "); int anInt = in.nextInt();

System.out.println("You entered " + anInt);

System.out.print("Enter a floating-point number: "); double aDouble = in.nextDouble();

System.out.println("You entered " + aDouble);

System.out.print("Enter 2 words: ");

String word1 = in.next(); // read a string delimited by white space

String word2 = in.next(); // read a string delimited by white space

System.out.println("You entered " + word1 + " " + word2);

in.nextLine(); // flush the "enter" before the next readLine()

System.out.print("Enter a line: ");

String line = in.nextLine(); // read a string up to line delimiter

System.out.println("You entered " + line);

}

}

### nextXxx() and hasNextXxx()

The Scanner class implements iterator<String> interface. You can use hasNext() coupled with next() to iterate through all the String tokens. You can also directly iterate through the primitive types via methods hasNextXxx() and nextXxx(). Xxx includes all primitive types (byte, short, int, long, float, double and Boolean), BigInteger, and BigNumber. char is not included but can be retrieved from String via charAt().

## Multithreading & Concurrent Programming



Java supports single-thread as well as multi-thread operations. A single-thread program has a single entry point (the main() method) and a singlex exit point. A multi-thread program has an initial entry point (the main() method), followed by many entry and exit points, which are run concurrently with the main(). The term "concurrency" refers to doing multiple tasks at the same time.

Java has built-in support for concurrent programming by running multiple threads concurrently within a single program.

### Process based Multitasking

* Executing multiple processes at the same time. The processes under execution are not dependent on each other. Every process will have its own set of resources.
* The process based multitasking is an operating systems approach
* Example: writing a java program, downloading a software, installing an application, playing audio etc.
* A multi-processing Operating System can run several processes at the same time
* Each process has its own address/memory space
* The OS's scheduler decides when each process is executed
* Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
* Separate processes do not have access to each other's memory space
* Many OSes have a shared memory system so that processes can share memory space

### Thread based Multitasking

* Executing different parts of the same program simultaneously is called Thread based Multitasking. The different parts (threads) of the program may(or may not) be dependent on each other. All the different parts(threads) of the program may share same resources.
* Thread based multitasking is also called Multithreading
* Multithreading is a process of executing different parts of an application that has multiple controls, where every

control can be considered as a “Thread”.

## Thread



* A Thread is a group of Statements that are executed separately. Thread is used for dividing the task of an application into separate sub-processes which can run simultaneously. A program can have multiple threads
* A thread consumes some resources so we should not use more thread than our requirement
* A Java threads shares the common memory area so memory allocation for each thread is not required. So context switching requires less time than using the multi-processing
* Every Java program has at least one thread i.e the thread that executes the Java Program (main thread)
* Threads are faster and more efficient program which increases speed of execution
* In a multithreaded application, there are several points of execution within the same memory space. - Each point of execution is called a thread
  + Threads share access to memory
* Example : OS will start the process which would run MS Power Point
  + We can consider the MS Power Point as main process, in order to make interactive execution, MS Power Point process can create multiple sub processes i.e thread
  + When we type anything in the slide then spell checker thread can be automatically created. We can also consider the auto-correction as thread while typing

### Multithreading Vs Multiprocessing

* OS can start multiple processes to achieve multitasking.
* In order to make execution faster, each process can be sub divided into the smaller chunk of lightweight subprocesses which is called as thread 

Key difference:

* Multithreading refers to an application with multiple threads running within a process
* Multiprocessing refers to an application organized across multiple OS-level processes

| **Thread** | **Process** |
| --- | --- |
| It is a programming concept in which a program or a process is divided into two or more subprograms or threads that are executed at the same time | It is an operating system approach in which multiple tasks are performed simultaneously |
| Threads does not requires separate address space | Process requires separate address space |
| Threads can be considered as sub process so they are lightweight | Processes are not lightweight. |
| Thread are not independent | Processes can be independent |
| Thread cannot be divided into multiple processes | Process can be divided into threads |

Java Program has At least one Thread:

* Each and every java program have at least on thread
* First thread is created when you invoke the static main method of your Java class ▪ Many java programs can have more than one thread which are created automatically.

| **Main Thread** |
| --- |

* Thread is mostly used for game development but non-game applications can also use multi-threading **Program No.**

| import java.lang.Thread; public class ThreadDemo\_157  { public static void main(String[] args)  {  Thread t = Thread.currentThread();  System.out.println(t);  }  } |
| --- |

**157**

Thread.currentThread() - it is a static method of java.lang.Thread Class which provides the information of current thread i.e. ThreadName, ThreadPriority, ThreadGroupName of the currently executing thread

A thread, also called a lightweightprocess, is a single sequential flow of programming operations, with a definite beginning and an end. During the lifetime of the thread, there is only a single point of execution. A thread by itself is not a program because it cannot run on its own. Instead, it runs within a program. The following figure shows a program with 3 threads running under a single CPU:



## Creating a new Thread



There are two ways to create a new thread:

### 1.By extending Thread Class

Extend a subclass from the superclass Thread and override the run() method to specify the running behavior of the thread. Create an instance and invoke the start() method, which will call-back the run() on a new thread. For example:

public class ThreadDemo

{

public static void main(String[] args)

{

Thread t1 = new Thread() {

// Create an instance of an anonymous inner class that extends Thread

public void run() {

// Override run() to specify the running behaviors

for (int i = 0; i < 20; ++i) { System.out.println("Thread 1 :" + i);

// provide the necessary delay try {

sleep(1000); // milliseconds } catch (InterruptedException ex) {}

}

}

};

t1.start(); // Start the thread. Call back run() in a new thread

Thread t2 = new Thread() {

public void run() { for (int i = 0; i < 20; ++i) { System.out.println("Thread 2 :" + i);

try {

sleep(1000);



class ClassA extends Thread

{

String name; public ClassA(String n)

{

name = n;

}

public void run()

{

for(int i=1; i<=5; i++)

{

System.out.println(name + " i = " + i);

}

System.out.println("Exit from " + name);

}

} public class ThreadDemo\_159

{

public static void main(String[] args)

{

* A newly created Thread will be in the born state or new state, then by calling the start() method thread moves from new state to runnable state
* start() method will call run() method which is overridden by the ThreadExample class.

### 2.By Implementing Runnable Interface

A Thread can be created by extending Thread class also. But Java allows only one class to extend, it wont allow multiple inheritance. So it is always better to create a thread by implementing Runnable interface. Java allows you to implement multiple interfaces at a time

By implementing Runnable interface, we need to provide implementation for run() method. To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method. 

Implementing Runnable interface does not create a Thread object, it only defines an entry point for threads in your object

It allows you to pass the object to the Thread(Runnable implementation) constructor

|  |  |  | System.out.println("obj1 = " + obj1.getState()); obj1.start();  ClassA obj2 = new ClassA("obj2"); obj2.start();  try  {  obj1.join(); obj2.join();  }catch(InterruptedException e){}  System.out.println("Exit from main"); |
| --- | --- | --- | --- |
| } | } |  |  |

Create a class that implements the Runnable interface and provide the implementation to the abstract method run() to specify the running behavior of the thread. Construct a newThread instance using the constructor with a Runnable object and invoke the start() method, which will call back run() on a new thread.

// Create an anonymous instance of an anonymous inner class that implements Runnable // and use the instance as the argument of Thread's constructor.

Thread t = new Thread(new Runnable() {

// Provide implementation to abstract method run() to specify the running behavior

@Override public void run() {

for (int i = 0; i < 100000; ++i) { if (stop) break;

tfCount.setText(count + "");

++count;

// Suspend itself and yield control to other threads

// Also provide the necessary delay

try {

Thread.sleep(10); // milliseconds

} catch (InterruptedException ex) {}

}

}

});

t.start(); // call back run() in new thread

The second method is needed as Java does not support multiple inheritance. If a class already extends from a certain superclass, it cannot extend from Thread, and have to implement theRunnable interface. The second method is also used to provide compatibility with JDK 1.1. It should be noted that the Thread class itself implements the Runnable interface.

The run() method specifies the running behavior of the thread and gives the thread something to do. You do not invoke the run() method directly from your program. Instead, you create aThread instance and invoke the start() method.

The start() method, in turn, will call back the run() on a new thread.

| **Implementing Runnable Interface And constructor** |
| --- |
| class MyThread implements Runnable  {  Thread t;  String tname;    public MyThread(String n)  {  tname = n; |

**161**

| t = new Thread(this, tname);  t.start();  }  public void run()  {  for(int i=1; i<=5; i++)  {  System.out.println(tname + "i = " + i);  }  System.out.println("\n\nExit from " + tname );  }  }  public class ThreadDemo\_161  {  public static void main(String[] args)  {  MyThread t1 = new MyThread("MyThread 1 ");  MyThread t2 = new MyThread("MyThread 2 ");  }  } |
| --- |



The class java.lang.Thread has the following constructors:

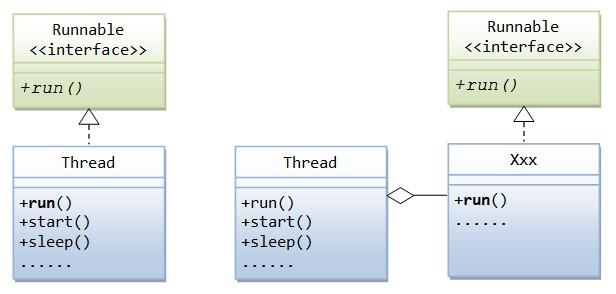
public **Thread**();

public **Thread**(String *threadName*); public **Thread**(Runnable *target*);

public **Thread**(Runnable *target*, String *threadName*);

The first two constructors are used for creating a thread by sub-classing the Thread class. The next two constructors are used for creating a thread with an instance of class that implements Runnable interface.

The class Thread implements Runnable interface, as shown in the class diagram.



As mentioned, the method run() specifies the running behavior of the thread. You do not invoke the run() method explicitly. Instead, you call the start() method of the class Thread. If a thread is constructed by extending the Thread class, the method start() will call back the overridden run() method in the extended class. On the other hand, if a thread is constructed by providing a Runnable object to the Thread's constructor, the start() method will call back the run() method of the Runnable object (and not the Thread's version).

Creating a new Thread by sub-classing Thread and overriding run() To create and run a new thread by extending Thread class:

1. Define a subclass (named or anonymous) that extends from the superclass Thread.
2. In the subclass, override the run() method to specify the thread's operations, (and provide other implementations such as constructors, variables and methods). 
3. A client class creates an instance of this new class. This instance is called a Runnable object (because Thread class itself implements Runnable interface).
4. The client class invokes the start() method of the Runnable object. The result is two thread running concurrently – the current thread continue after invoking the start(), and a new thread that executes run() method of the Runnable object.

For example,

class MyThread extends Thread { // override the run() method

@Override public void run() {

// Thread's running behavior

}

// constructors, other variables and methods ...... }

public class Client {

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

// Start a new thread MyThread t1 = new MyThread(); t1.start(); // Called back run() ......

// Start another thread new MyThread().start(); ......

}

}

Often, an inner class (named or anonymous) is used instead of a ordinary subclass. This is done for readability and for providing access to the private variables and methods of the outer class. For example,

public class Client {



...

// You can also used a named inner class defined below new MyThread().start();

}

// Define a named inner class extends Thread

class MyThread **extends Thread** { public void **run**() {

// Thread's running behavior

// Can access the private variables and methods of the outer class

}

}

}

Example public class MyThread extends Thread { private String name;

public MyThread(String name) { // constructor this.name = name;

}

// Override the run() method to specify the thread's running behavior

@Override public void run() {

for (int i = 1; i <= 5; ++i) {

System.out.println(name + ": " + i); yield();

}

} }

A class called MyThead is created by extending Thread class and overriding the run() method. A constructor is defined to takes a String as the name of the thread. The run() method prints 1 to 5, but invokes yield() to yield control to



for (Thread t : threads) {

t.start();

}

}

}

Take note that the output is indeterminate (different run is likely to produce different output), as we do not have complete control on how the threads would be executed.

### Creating a new Thread by implementing the Runnable Interface

To create and run a new thread by implementing Runnable interface:

1. Define a class that implements the Runnable interface.
2. In the class, provide implementation to the abstract method run() to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).
3. A client class creates an instance of this new class. The instance is called a Runnable object.
4. The client class then constructs a new Thread object with the Runnable object as argument to the constructor, and invokes the start() method. The start() called back the run() in the Runnable object (instead of the Thread class).

class MyRunnable extends SomeClass **implements Runnable** {

// provide implementation to abstract method run()

@Override public void **run**() {

// Thread's running behavior

}

......

// constructors, other variables and methods

}

... 

}

public void run() {

// Thread's running behavior

// Can access the private variables and methods of the outer class

}

});

t.start();

## Methods in the Thread Class



The methods available in Thread class include:

public void start(): Begin a new thread. JRE calls back the run() method of this class. The current thread continues. public void run(): to specify the execution flow of the new thread. When run() completes, the thread terminates.

public static sleep(long millis) public static sleep(long millis, int nanos) public void interrupt()

Suspend the current thread and yield control to other threads for the given milliseconds (plus nanoseconds).

Method sleep() is thread-safe as it does not release its monitors. You can awaken a sleep thread before the specified timing via a call to the interrupt () method. The awaken thread will throw an InterruptedException and execute its InterruptedException handler before resuming its operation. This is a static method (which does not require an instance) and commonly used to pause the current thread (via Thread.sleep()) so that the other threads can have a chance to execute. It also provides the necessary delay in many applications. For example: try {

// Suspend the current thread and give other threads a chance to run

// Also provide the necessary delay

Thread.sleep(100); // milliseconds

} catch (InterruptedException ex) {}

public static void yield(): hint to the scheduler that the current thread is willing to yield its current use of a processor to allow other threads to run. The scheduler is, however, free to ignore this hint. Rarely-used.

public boolean isAlive(): Return false if the thread is new or dead. Returns true if the thread is "runnable" or "not runnable".

public void setPriority(int p): Set the priority-level of the thread, which is implementation dependent.

The stop(), suspend(), and resume() methods have been deprecated in JDK 1.4, because they are not thread-safe, due to the release of monitors. See JDK API documentation for more discussion. 

**Program No.**

**Thread Example Tortoise and hair story** 

class Racer implements Runnable

{

private static String Winner; public Thread t;

public String racername;

public Racer(String rn)

{

racername = rn;

t = new Thread(this, racername);

t.start();

}

public void run()

{

startrace();

}

public void startrace()

{

for(int distance = 1; distance<=100; distance++)

{

System.out.println(racername + " covered : "

+distance + " meters");

if(isRaceWon) 

{

break;

}

}

}

public boolean isWon(int distance)

{

boolean result;

if(Racer.Winner==null && distance==100)

{

//declare winner

Racer.Winner = racername;

System.out.println("\n\n\t Race Winner is " +

Racer.Winner);

result = true;

}

else if(Racer.Winner==null)

{

result = false;

}

}

public class ThreadDemo\_162

{

public static void main(String[] args)

{

Racer tortoise = new Racer("Tortoise");

Racer hair = new Racer("Hair");

}

How to pause a thread:



You can make the currently running thread pauses its execution by invoking the static method sleep(milliseconds) of the Thread class. Then the current thread is put into sleeping state. Here’s how to pause the current thread:

try {

Thread.sleep(2000);

} catch (InterruptedException ex) {

}

This code pauses the current thread for about 2 seconds (or 2000 milliseconds). After that amount of time, the thread returns to continue running normally.

InterruptedException is a checked exception so you must handle it. This exception is thrown when the thread is interrupted by another thread.

Let’s see a full example. The following NumberPrint program is updated to print 5 numbers, each after every 2 seconds:

| **DEESHA COMPUTER EDUCATION** | | **[ [ JAVA – CORE ]** | |
| --- | --- | --- | --- |
|  | // code to resume or terminate... | |  |

public class NumberPrint implements Runnable { public void run() { for (int i = 1; i <= 5; i++) {



public static void main(String[] args) { Runnable task = new NumberPrint(); Thread thread = new Thread(task); thread.start();

}

}

Note that you can’t pause a thread from another thread. Only the thread itself can pause its execution. And there’s no guarantee that the thread always sleep exactly for the specified time because it can be interrupted by another thread, which is described in the next section.

\* How to interrupt a thread:



Interrupting a thread can be used to stop or resume the execution of that thread from another thread. For example, the following statement interrupts the thread t1 from the current thread:

t1.interrupt();

If t1 is sleeping, then calling interrupt() on t1 will cause the InterruptedException to be thrown. And whether the thread should stop or resume depending on the handling code in the catch block.

In the following code example, the thread t1 prints a message after every 2 seconds, and the main thread interrupts t1 after 5 seconds:

public class ThreadInterruptExample implements Runnable { public void run() {



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

Thread t1 = new Thread(new ThreadInterruptExample()); t1.start(); try {

Thread.sleep(5000); t1.interrupt();

} catch (InterruptedException ex) {

}

}

}

As you can see in the catch block in the run() method, it continues the for loop when the thread is interrupted:

try {

Thread.sleep(2000);

| **DEESHA COMPUTER EDUCATION** | | **[ [ JAVA – CORE ]** | |
| --- | --- | --- | --- |
|  | // do nothing |  |  |

} catch (InterruptedException ex) { System.out.println("I'm resumed");



return;

}

You see, the return statement causes the run() method to return which means the thread terminates and goes to dead state.

What if a thread doesn’t sleep (no handling for InterruptedException)?

In such case, you need to check the interrupt status of the current thread using either of the following methods of the Thread class:

* interrupted(): this static method returns true if the current thread has been interrupted, or false otherwise. Note that this method clears the interrupt status, meaning that if it returns true, then the interrupt status is set to false.
* isInterrupted(): this non-static method checks the interrupt status of the current thread and it doesn’t clear the interrupt status.

The ThreadInterruptExample above can be modified to use the checking method as below:

public class ThreadInterruptExample implements Runnable { public void run() { for (int i = 1; i <= 10; i++) {

System.out.println("This is message #" + i); if (Thread.interrupted()) {

System.out.println("I'm about to stop"); return;

}



} catch (InterruptedException ex) {

// do nothing

}

}

}

However this version doesn’t behave the same as the previous one because the thread t1 terminates very quickly as it doesn’t sleep and the print statements are executed very fast. So this example is just to show you how it is used. In practice, this kind of checking on interrupt status should be applied for long-running operations such as IO, network, database, etc.

And remember that when the InterruptedException is thrown, the interrupt status is cleared.

If you look at the Thread class in Javadocs, you will see there are 4 methods:

However all these methods are deprecated, meaning that you shouldn’t use them. Let use the interruption mechanism I have described so far.

\* How to make a thread waits other threads?



This is called joining and is useful in case you want the current thread to wait for other threads to complete. After that the current thread continues running. For example: t1.join();

This statement causes the current thread to wait for the thread t1 to complete before it continues. In the following program, the current thread (main) waits for the thread t1to complete:

| **DEESHA COMPUTER EDUCATION** | | **[ [ JAVA – CORE ]** | |
| --- | --- | --- | --- |
|  | destroy() - stop() - suspend() - resume() | |  |

public class ThreadJoinExample implements Runnable {



return;

}

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

Thread t1 = new Thread(new ThreadJoinExample()); t1.start(); try { t1.join();

| } catch (InterruptedException ex) { |  |  |
| --- | --- | --- |
| **www.deeshanotes.com** | **VISHAL** | 285 |

}

System.out.println("I'm " + Thread.currentThread().getName()); }

}

In this program, the current thread (main) always terminates after the thread t1 completes. Hence you see the message “I’m main” is always printed last:

This is message #1

This is message #2

| **DEESHA COMPUTER EDUCATION** | | **[ [ JAVA – CORE ]** | |
| --- | --- | --- | --- |
|  | // do nothing |  |  |

This is message #3



Note that the join() method throws InterruptedException if the current thread is interrupted, so you need to catch it.

There are 2 overloads of join() method:

* join(milliseconds)
* join(milliseconds, nanoseconds)

These methods cause the current thread to wait at most for the specified time. That means if the time expires and the joined thread has not completed, the current thread continues running normally.

You can also join multiple threads with the current thread, for example:

t1.join(); t2.join();

| **DEESHA COMPUTER EDUCATION** | | **[ [ JAVA – CORE ]** | |
| --- | --- | --- | --- |
|  | t3.join(); |  |  |

In this case, the current thread has to wait for all three threads t1, t2 and t3 completes before it can resume running.



## The Life Cycle of a Thread

The thread is in the "new" state, once it is constructed. In this state, it is merely an object in the heap, without any system resources allocated for execution. From the "new" state, the only thing you can do is to invoke the start() method, which puts the thread into the "runnable" state. Calling any method besides the start() will trigger an IllegalThreadStateException. 

The start() method allocates the system resources necessary to execute the thread, schedules the thread to be run, and calls back the run() once it is scheduled. This put the thread into the "runnable" state. However, most computers have a single CPU and *time-slice* the CPU to support multithreading. Hence, in the "runnable" state, the thread may be running or waiting for its turn of the CPU time.

A thread cannot be started twice, which triggers a runtime IllegalThreadStateException.

The thread enters the "not-runnable" state when one of these events occurs:

1. The sleep() method is called to suspend the thread for a specified amount of time to yield control to the other threads. You can also invoke the yield() to hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is, however, free to ignore this hint.
2. The wait() method is called to wait for a specific condition to be satisfied.
3. The thread is *blocked* and waiting for an I/O operation to be completed.

For the "non-runnable" state, the thread becomes "runnable" again:

1. If the thread was put to sleep, the specified sleep-time expired or the sleep was interrupted via a call to the interrupt() method.
2. If the thread was put to wait via wait(), its notify() or notifyAll() method was invoked to inform the waiting thread that the specified condition had been fulfilled and the wait was over.
3. If the thread was blocked for an I/O operation, the I/O operation has been completed.

A thread is in a "terminated" state, only when the run() method terminates naturally and exits.

The method isAlive() can be used to test whether the thread is alive. The isAlive() returns false if the thread is "new" or "terminated". It returns true if the thread is "runnable" or "not-runnable".

## Thread States (Thread Life Cycle) in Java



A thread can go through various states during its life. The Thread’s getState() method returns an enum constant that indicates current state of the thread, which falls in one of the following values:

* RUNNABLE
* BLOCKED
* WAITING
* TIMED\_WAITING
* TERMINATED

These enum constants are defined in the Thread.State enum. Let me explain each state in details.

* **NEW**: when a thread is created but has not executed (the start() method has not been invoked), it is in the new state.

* **RUNNABLE**: when the start() method has been invoked, the thread enters the runnable state, and its run() method is executing. Note that the thread can come back to runnable state from another state (waiting, blocked), but it may not be picked immediately by the thread scheduler, hence the term “runnable”, not running.

* **BLOCKED**: when a thread tries to acquire an intrinsic lock (not a lock in the java.util.concurrent package) that is currently held by another thread, it becomes blocked. When all other threads have relinquished the lock and the thread scheduler has allowed this thread to hold the lock, the thread becomes unblocked and enters the runnable state. 

* **WAITING**: a thread enters this state if it waits to be notified by another thread, which is the result of calling

Object.wait() or Thread.join(). The thread also enters waiting state if it waits for a Lock or Condition in the java.util.concurrent package. When another thread calls Object‘s notify()/notifyAll() or Condition’s signal()/signalAll(), the thread comes back to the runnable state.

* **TIMED\_WAITING**: a thread enters this state if a method with timeout parameter is called: sleep(), wait(), join(), Lock.tryLock() and Condition.await(). The thread exits this state if the timeout expires or the appropriate notification has been received.

* **TERMINATED**: a thread enters terminated state when it has completed execution. The thread terminates for one of two reasons:

+ the run() method exits normally.

+ the run() method exits abruptly due to a uncaught exception occurs.

The following diagram helps you visually understand the thread states and transitions between them:

And the following code example illustrates how to check state of a thread:

public class ThreadState {

public static void main(String[] args) throws InterruptedException {

Thread t = new Thread(new Runnable() {

public void run() {

Thread self = Thread.currentThread();

System.out.println(self.getName() + "is " + self.getState());// LINE 0

}

});

System.out.println(t.getName() + "is " + t.getState()); // LINE 1

t.start();

t.join(); if (t.getState() == Thread.State.TERMINATED) {

System.out.println(t.getName() + " is terminated"); // LINE 2

} 

}

}

Line 1 prints Thread-0 is NEW because at this time the thread t has not been started.

When the thread started, its run method execute, hence line 0 prints Thread-0 is RUNNABLE.

The call t.join() causes the main thread to wait for the thread t to finish, hence line 2 is always executed after thread t completes, thus the output Thread-0 is terminated.

Note that the thread’s state may change after the call to getState(). That means calling getState() may not reflect the actual state of the thread only a moment later.

| **Thread Phases** |
| --- |
| import java.lang.Thread;    public class ThreadDemo extends Thread  {    public void run()  {  System.out.println("Thread is running !!");  }  public static void main(String[] args){ ThreadDemo t1 = new ThreadDemo(); |

**Program No. 158**

| } | ThreadDemo t2 = new ThreadDemo();  System.out.println("T1 ==> " + t1.getState());  System.out.println("T2 ==> " + t2.getState());  t1.start();  System.out.println("T1 ==> " + t1.getState());  System.out.println("T2 ==> " + t2.getState());  t2.start();  System.out.println("T1 ==> " + t1.getState());  System.out.println("T2 ==> " + t2.getState()); } |
| --- | --- |

## Thread Scheduling and Priority

JVM implements a fixed priority thread-scheduling scheme. Each thread is assigned a priority number (between the Thread.MIN\_PRIORITY and Thread.MAX\_PRIORITY). The higher the number, the higher is the priority for the thread. When a new thread is created, it inherits the priority number from the thread that created it. You can used the method setPriority() to change the priority number of a thread as follows: 

public void setPriority(int priority);

The int priority is JVM dependent. It may take a value between 1 (lowest priority) to 10.

JVM chooses the highest-priority thread for execution. If there is more than one thread with the same highest-priority, JVM schedules them in a round-robin manner.

JVM also implements a preemptive scheduling scheme. In a preemptive environment, if at any time a higher priority thread becomes "runnable", the current lower priority thread will yield control to the higher priority thread immediately.

If there are more than one equal-priority runnable threads, one thread may run until the completion without yielding control to other equal-priority threads. This is known as *starvation*. Therefore, it is a good practice to yield control to other equal-priority thread via the sleep() or yield() method. However, you can never yield control to a lowerpriority thread.

In some operating systems such as Windows, each of the running thread is given a specific amount of CPU time. It is known as time slicing to prevent a thread from starving the other equal-priority threads. However, do not rely on time slicing, as it is implementation dependent.

Hence, a running thread will continue running until:

* A higher priority thread becomes "runnable".
* The running thread yields control voluntarily by calling methods such as sleep(), yield(), and wait().
* The running thread terminates, i.e., its run() method exits.
* On system that implements time slicing, the running thread consumes its CPU time quota.

An important point to note is the thread scheduling and priority is JVM dependent. This is natural as JVM is a virtual machine and requires the native operating system resources to support multithreading. Most JVM does not guarantee that the highest-priority thread is being run at all times. It may choose to dispatch a lower-priority thread for some reasons such as to prevent starvation. Therefore, you should not rely on the priority in your algorithm.

## Monitor Lock & Synchronization



A monitor is an object that can be used to block and revive thread. It is supported in the java.lang.Object root class, via these mechanisms:

1. A lock for each object.
2. The keyword synchronized for accessing object's lock.
3. The wait(), notify() and notifyAll() methods in java.lang.Object for controlling threads.

Each Java object has a lock. At any time, the lock is controlled by, at most, a single thread. You could mark a method or a block of the codes with keyword synchronized. A thread that wants to execute an object's synchronized code must first attempt to acquire its lock. If the lock is under the control of another thread, then the attempting thread goes into the *Seeking Lock* state and becomes ready only when the lock becomes available. When a thread that owns a lock completes the synchronized code, it gives up the lock.

## Keyword "synchronized"

synchronized(anObject) { // synchronized a block of codes based on another object ...... } ...... }

Synchronization can be controlled at method level or block level. Variables cannot be synchronized. You need to synchronized the ALL the methods that access the variables.

private static int counter = 0;

public static synchronized void increment() {

++counter;

}

public static synchronized void decrement() {

--counter; }

You can also synchronized on static methods. In this case, the *class lock* (instead of the instance lock) needs to be acquired in order to execute the method.

Example public class SynchronizedCounter { private static int count = 0;

public synchronized static void increment() {

++count;

System.out.println("Count is " + count + " @ " + System.nanoTime());

}

public synchronized static void decrement() {

--count;

System.out.println("Count is " + count + " @ " + System.nanoTime()); }



} catch (InterruptedException e) {}

}

}

};

Thread threadDecrement = new Thread() {

@Override public void run() {

for (int i = 0; i < 10; ++i) { SynchronizedCounter.decrement(); try {

sleep(1);

} catch (InterruptedException e) {}

}

}

};

threadIncrement.start(); threadDecrement.start();

}

}

It is important to note that while the object is locked, synchronized methods and codes are blocked. However, nonsynchronized methods can proceed without acquiring the lock. Hence, it is necessary to synchronize all the methods involved the shared resources. For example, if synchronized access to a variable is desired, all the methods to that variable should besynchronized. Otherwise, a non-synchronized method can proceed without first obtaining the lock, which may corrupt the state of the variable.

## wait(), notify() & notifyAll() for Inter-Thread Synchronization

These methods are defined in the java.lang.Object class (instead of java.land.Thread class). These methods can only be called in the *synchronous* codes.

The wait() and notify() methods provide a way for a shared object to pause a thread when it becomes unavailable to that thread and to allow the thread to continue when appropriate.

### Example: Consumer and Producer

In this example, a producer produces a message (via putMessage() method) that is to be consumed by the consumer (via getMessage() method), before it can produce the next message. In a so-called producer-consumer pattern, one thread can suspend itself using wait() (and release the lock) until such time when another thread awaken it using notify() or notifyAll().

// Testing wait() and notify() public class MessageBox { private String message; private boolean hasMessage;

// producer

public synchronized void putMessage(String message) {

while (hasMessage) { // no room for new message try {

wait(); // release the lock of this object

} catch (InterruptedException e) { }

}

// acquire the lock and continue hasMessage = true;

this.message = message + " Put @ " + System.nanoTime(); notify();

}



hasMessage = false; notify();

return message + " Get @ " + System.nanoTime();

} }

public class TestMessageBox { public static void main(String[] args) { final MessageBox box = new MessageBox();

Thread producerThread = new Thread() {

@Override public void run() {

System.out.println("Producer thread started..."); for (int i = 1; i <= 6; ++i) { box.putMessage("message " + i);

System.out.println("Put message " + i);

}

}

};

Thread consumerThread1 = new Thread() {

@Override public void run() {

System.out.println("Consumer thread 1 started..."); for (int i = 1; i <= 3; ++i) {

System.out.println("Consumer thread 1 Get " + box.getMessage());

} 

}

};

System.out.println("Consumer thread 2 started..."); for (int i = 1; i <= 3; ++i) {

System.out.println("Consumer thread 2 Get " + box.getMessage());

}

}

};

consumerThread1.start(); consumerThread2.start(); producerThread.start();

}

}

The output messages (on System.out) may appear out-of-order. But closer inspection on the put/get timestamp confirms the correct sequence of operations.

The synchronized producer method putMessage() acquires the lock of this object, check if the previous message has been cleared. Otherwise, it calls wait(), releases the lock of this object, goes into WAITING state and places this thread on this object's "wait" set. On the other hand, the synchronized consumer's method getMessage() acquires the lock of this object and checks for new message. If there is a new message, it clears the message and issues notify(), which arbitrarily picks a thread on this object's "wait" set (which happens to be the producer thread in this case) and place it on BLOCKED state. The consumer thread, in turn, goes into the WAITING state and placed itself in the "wait" set of this object (after the wait() method). The producer thread then acquires the thread and continue its operations.

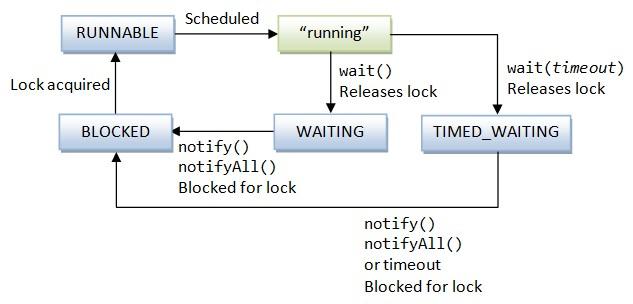


### wait() with timeout

There are variations of wait() which takes in a timeout value:

public final void wait() throws InterruptedException public final void wait(long timeout) throws InterruptedException public final void wait(long timeout, int nanos) throws InterruptedException

The thread will ALSO go to BLOCKED state after the timeout expired.



| **Interthread communication** |
| --- |
|  |

**Program No.**

**158**





## Deadlock



**Deadlock** describes a situation where two more threads are blocked because of waiting for each other forever. When deadlock occurs, the program hangs forever and the only thing you can do is to kill the program

The program encounters a deadlock and cannot continue

public class Business {

private Object lock1 = new Object(); private Object lock2 = new Object(); public void foo() { synchronized (lock1) { synchronized (lock2) {

System.out.println("foo");

}

} }

public void bar() { synchronized (lock2) { synchronized (lock1) {

System.out.println("bar");

}

}

} }

As you can see, both the methods foo() and bar() try to acquire two lock objects lock1 and lock2 but in different order.

And consider the following test program:

public class BusinessTest1 {

public static void main(String[] args) { Business business = new Business(); Thread t1 = new Thread(new Runnable() { public void run() {

} }); t2.start(); 

} }

shared instance of the Business class. But deadlock is likely never to occur because one thread can execute and exit a method very quickly so the other thread have chance to acquire the locks.

Let’s modify this test program in order to create 10 threads for executing foo() and other 10 threads for executing bar() as follows:

public class BusinessTest2 {

public static void main(String[] args) { Business business = new Business(); for (int i = 0; i < 10; i++) { new Thread(new Runnable() { public void run() { business.foo();

}

}).start();

}

for (int i = 0; i < 10; i++) { new Thread(new Runnable() { public void run() { business.bar();

}

}).start();

}

}

}

Run this program several times (4-10 times), you will see that sometimes the program runs fine:



Why? It’s because deadlock happens. Let me explain how:

* Thread 1 enters foo() method and it acquires lock1. At the same time, thread 2 enters bar() method and it acquires lock2.
* Thread 1 tries to acquire lock2 which is currently held by thread 2, hence thread 1 blocks.
* Thread 2 tries to acquire lock1 which is currently held by thread 1, hence thread 2 blocks.

Both threads block each other forever, deadlock occurs and the program hangs.

So how to avoid deadlock?

Java doesn’t have anything to escape deadlock state when it occurs, so you have to design your program to avoid deadlock situation. Avoid acquiring more than one lock at a time. If not, make sure that you acquire multiple locks in consistent order. In the above example, you can avoid deadlock by synchronize two locks in the same order in both methods:

public void foo() { synchronized (lock1) { synchronized (lock2) {

System.out.println("foo");

}

}

}

public void bar() { synchronized (lock1) { synchronized (lock2) {

System.out.println("bar");

}

}

}

Also try to shrink the synchronized blocks as small as possible to avoid unnecessary locking on code that doesn’t need to be synchronized.



s