##### **1-Features of Java**

##### Features of a language are nothing but the set of services or facilities provided by the language vendors to the industry programmers. Some important **features of java** are;

##### features of java

##### **1. Simple:-**

##### **It is simple because of the following factors:**

##### It is **free from pointer** due to this execution time of application is improve. [whenever we write a Java program without pointers then internally it is converted into the equivalent pointer program].

##### It have **Rich set of API** (application protocol interface).

##### It have **Garbage Collector** which is always used to collect un-Referenced (unused) Memory location for improving performance of a Java program.

##### It contains user friendly syntax for developing any applications.

##### simple

##### **2. Platform Independent:-**

##### A program or technology is said to be platform independent if and only if which can run on all available operating systems with respect to its development and compilation. (Platform represents O.S).

##### platform independent

##### **3. Architectural Neutral**

##### Architecture represents processor. A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering there architecture and vendor (providers) irrespect to its development and compilation.

##### archetecture neutral

##### The languages like C, CPP are treated as architectural dependent.

##### **4. Portable**

##### If any language supports platform independent and architectural neutral feature known as portable. The languages like C, CPP, Pascal are treated as non-portable language. It is a portable language. According to SUN microsystem.

##### Portable

##### **5. Multithreaded**

##### A flow of control is known as thread. When any Language execute multiple thread at a time that language is known as multithreaded Language. It is multithreaded Language.

##### **6. Distributed**

##### Using this language we can create distributed application. RMI and EJB are used for creating distributed applications. In distributed application multiple client system are depends on multiple server systems so that even problem occurred in one server will never be reflected on any client system.

##### Distributed Application

##### **Note:**In this architecture same application is distributed in multiple server system.

##### **7. Networked**

##### It is mainly design for web based applications, J2EE is used for developing network based applications.

##### **8. Robust**

##### Simply means of Robust is strong. It is robust or strong Programming Language because of its capability to handle Run-time Error, automatic garbage collection, lack of pointer concept, Exception Handling. All these points makes It robust Language.

##### **9. Dynamic**

##### It support Dynamic memory allocation due to this memory wastage is reduce and improve performance of application. The process of allocating the memory space to the input of the program at a run-time is known as dynamic memory allocation, To programming to allocate memory space by dynamically we use an operator called 'new' 'new' operator is known as dynamic memory allocation operator.

##### **10. Secure**

##### It is more secured language compare to other language; In this language all code is covered into byte code after compilation which is not readable by human.

##### **11. High performance**

##### It have high performance because of following reasons;

##### This language **uses Bytecode** which is more faster than ordinary pointer code so Performance of this language is high.

##### **Garbage collector**, collect the unused memory space and improve the performance of application.

##### It have **no pointers** so that using this language we can develop an application very easily.

##### It **support multithreading**, because of this time consuming process can be reduced to execute the program.

##### **12. Interpreted**

##### It is one of the highly interpreted programming languages.

##### **13. Object Oriented**

##### It supports OOP's concepts because of this it is most secure language, for this topic you can read our oop's concepts in detail.

##### **2-Java - Basic Program**

##### **Requirements for java Program**

##### For executing any java program we need given things.

##### Install the JDK any version if you don't have installed it.

##### Set path of the jdk/bin directory.

##### Create the java program

##### Compile and run the java program

**Programs**

class First

{

public static void main(String[] args)

{

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

System.out.println("My First Java Program");

}

}

##### **Explain Public static void main (String args[])**

##### **Public-**It means that you can call this method from outside of the class you are currently in. This is necessary because this method is being called by the Java runtime system which is not located in you current class.

##### **Static-**When the JVM makes call to the main method there is no object existing for the class being called therefore it has to have static method to allow invocation from class.

##### **Void-**Java is platform independent language and if it will return some value then the value may mean different things to different platforms.

##### **Main-**It's just the name of method. This name is fixed and as it's called by the JVM as entry point for an application.

##### **String args[]-**These are the arguments of type String that your Java application accepts when you run it.

**Difference between JDK, JVM and JRE**

##### Jvm, Jre, Jdk these all the backbone of java language. Each components have separate works. Jdk and Jre physically exists but Jvm are abstract machine it means it not physically exists.

##### **JVM-**JVM (Java Virtual Machine) is a software. It is a specification that provides runtime environment in which java bytecode can be executed. It not physically exists.

##### JVMs are not same for all hardware and software, for example for window os JVM is different and for Linux VJM is different. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

##### **JRE-**The Java Runtime Environment (JRE) is part of the Java Development Kit (JDK). It contains set of libraries and tools for developing java application. The Java Runtime Environment provides the minimum requirements for executing a Java application. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

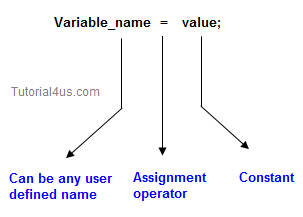
##### jre

##### **JDK-**The Java Development Kit (JDK) is primary components. It physically exists. It is collection of programming tools and JRE, JVM.

##### jkd

##### **3-Variables and Data Type in Java**

**Variable** is an identifier which holds data or another one variable is an identifier whose value can be changed at the execution time of program. Variable is an identifier which can be used to identify input data in a program.



## Syntax

Variable\_name = value;

**Rules to declare a Variable**

* Every variable name should start with either alphabets or underscore ( \_ ) or dollar ( $ ) symbol.
* No space are allowed in the variable declarations.
* Except underscore ( \_ ) no special symbol are allowed in the middle of variable declaration
* Variable name always should exist in the left hand side of assignment operators.
* Maximum length of variable is 64 characters.
* No keywords should access variable name.

**Note:** Actually a variable also can start with ¥,¢, or any other currency sign.

## Example of Variable Declaration

class Sum

{

public static void main(String[] args)

{

int \_a, ¢b, ¥c, $d, result;

\_a=10;

¢b=20;

¥c=30;

$d=40;

result=\_a+¢b+¥c+$d;

System.out.println("Sum is :" +result);

}

}

## Output

Sum is : 100

**Variable declarations-** In which sufficient memory will be allocated and holds default values.

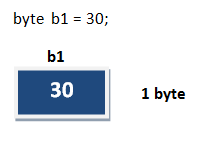
Syntax

Datatype variable\_name;

byte b1;

## Variable in java

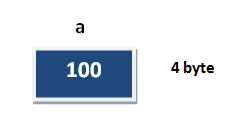
## Variable initialization-It is the process of storing user defined values at the time of allocation of memory space.



## Variable assignment-Value is assigned to a variable if that is already declared or initialized.

Variable\_Name = value

int a = 100;



## Syntax

int a= 100;

int b;

b = 25; // ------> direct assigned variable

b = a; // ------> assigned value in term of variable

b = a+15; // ------> assigned value as term of expression

**Variable**

Variable is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "**vary + able**" that means its value can be changed.

**Types of Variables**

1. **Local Variable**- A variable which is declared inside the method is called local variable.
2. **Instance Variable**- A variable which is declared inside the class but outside the method, is called instance variable. It is not declared as static.
3. **Static variable**-A variable that is declared as static is called static variable. It cannot be local.

**Data-Type**

Java language has a rich implementation of data types. Data types specify size and the type of values that can be stored in an identifier.



* **Primitive Data type**
* **Non-Primitive Data type**

**Primitive Data Types:**

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

**byte:**

* Byte data type is a 8-bit signed two's complement integer.
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
* Example : byte a = 100 , byte b = -50

**short:**

* Short data type is a 16-bit signed two's complement integer.
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767(inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
* Default value is 0.
* Example : short s= 10000 , short r = -20000

**int:**

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648.(-2^31)
* Maximum value is 2,147,483,647(inclusive).(2^31 -1)
* Int is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0.
* Example : int a = 100000, int b = -200000

**long:**

* Long data type is a 64-bit signed two's complement integer.
* Minimum value is -9,223,372,036,854,775,808.(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
* This type is used when a wider range than int is needed.
* Default value is 0L.
* Example : long a = 100000L, int b = -200000L

**float:**

* Float data type is a single-precision 32-bit IEEE 754 floating point.
* Float is mainly used to save memory in large arrays of floating point numbers.
* Default value is 0.0f.
* Float data type is never used for precise values such as currency.
* Example : float f1 = 234.5f

**double:**

* double data type is a double-precision 64-bit IEEE 754 floating point.
* This data type is generally used as the default data type for decimal values. generally the default choice.
* Double data type should never be used for precise values such as currency.
* Default value is 0.0d.
* Example : double d1 = 123.4

**boolean:**

* boolean data type represents one bit of information.
* There are only two possible values : true and false.
* This data type is used for simple flags that track true/false conditions.
* Default value is false.
* Example : boolean one = true

**char:**

* char data type is a single 16-bit Unicode character.
* Minimum value is '\u0000' (or 0).
* Maximum value is '\uffff' (or 65,535 inclusive).
* Char data type is used to store any character.
* Example . char letterA ='A'

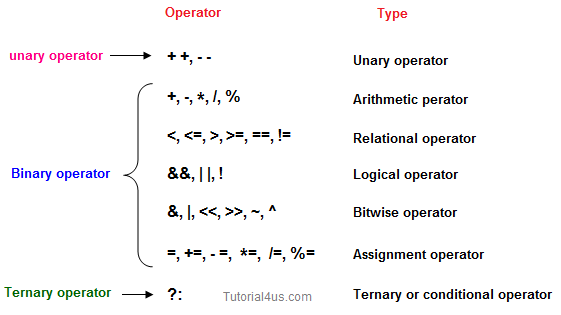
**Non-Primitive (Reference) Data type**

A reference data type is used to refer to an object. A reference variable is declare to be of specific and that type can never be change. We will talk a lot more about reference data type later in Classes and Object lesson.

##### **4-Operators in Java**

Operator is a special symbol that tells the compiler to perform specific mathematical or logical Operation. Java supports following lists of operators.

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Ternary or Conditional Operators



**Arithmetic Operators-** Given table shows all the Arithmetic operator supported by Java Language. Lets suppose variable **A** hold 8 and **B** hold 3.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example (int A=8, B=3)** | **Result** |
| + | A+B | 11 |
| - | A-B | 5 |
| \* | A\*B | 24 |
| / | A/B | 2 |
| % | A%4 | 0 |

Arithmetic operators are used in mathematical like addition, subtraction etc. The following table lists the arithmetic operators:

Assume that int X = 10 and int Y = 20

| **Operators** | **Description** |
| --- | --- |
| + | Addition – Adds values on either side of the operator |
| - | Subtraction – Subtracts right hand operand from left hand operand |
| \* | Multiplication – Multiplies values on either side of the operand |
| / | Division - Divides left hand operand by right hand operand |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder |
| ++ | Increment - Increase the value of operand by 1 |
| -- | Decrement - Decrease the value of operand by 1 |

|  |
| --- |
|  |

Example:

public class Main{

public static void main(String args[]{

Int X = 10;

Int Y = 20;

System.out.println("Addition (X+Y) = "+(X+Y)); // return 30

System.out.println("Subtraction (X-Y) = "+(X-Y)); // return -10

System.out.println("Multiplication (X\*Y) = "+(X\*Y)); // return 200

System.out.println("Division (Y/X) = "+(Y/X)); // return 2

System.out.println("Addition (Y%X) = "+(Y%X)); // return 0

Y++;

System.out.println("Increment Y = "+Y); // return 21

X--;

System.out.println("Decrement X = "+X); // return 9}}

**Relational Operators**

Which can be used to check the Condition, it always return true or false. Lets suppose variable **A** hold 8 and **B** hold 3.

|  |  |  |
| --- | --- | --- |
| Operators | Example (int A=8, B=3) | Result |
| < | A<B | False |
| <= | A<=10 | True |
| > | A>B | True |
| >= | A<=B | False |
| == | A== B | False |
| != | A!=(-4) | True |

There are following relational operators supported by Java language like ==, ! = etc.

Assume variable X=10 and variable Y=20 then:

| **Operator** | **Description** |
| --- | --- |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. |

**Example :**

public class Main{

public static void main(String args[]){

int X = 10;

int Y = 20;

System.out.println("(X == Y) = "+(X == Y));

System.out.println("(X != Y) = "+(X != Y));

System.out.println("(X > Y) = "+(X > Y));

System.out.println("(X < Y) = "+(X < Y));

System.out.println("(X >= Y) = "+(X >= Y));

System.out.println("(X <= Y) = "+(X <= Y)); }}

**Logical Operator**

Which can be used to combine more than one Condition?. Suppose you want to combined two conditions **A<B** and **B>C**, then you need to use **Logical Operator** like (A<B) && (B>C). Here **&&**is Logical Operator.

|  |  |  |
| --- | --- | --- |
| Operator | Example (int A=8, B=3, C=-10) | Result |
| && | (A<B) && (B>C) | False |
| || | (B!=-C) || (A==B) | True |
| ! | !(B<=-A) | True |

Truth table of Logical Operator

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C1 | C2 | C1 && C2 | C1 || C2 | !C1 | !C2 |
| T | T | T | T | F | F |
| T | F | F | T | F | T |
| F | T | F | T | T | F |
| F | F | F | F | T | T |

The following table lists the logical operators like &&, || etc. This logical operator use for join two conditions.

Assume boolean variables X=true and variable Y=false then:

| **Operator** | **Description** |
| --- | --- |
| && | Called Logical AND operator. If both the operands are non zero then then condition becomes true. |
| || | Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. |

Example:

public class Main{

public static void main(String args[]){

int X = 60;

int Y = 13;

if((X == Y) && (X != Y)){

System.out.println("True");

}else{

System.out.println("False");

}

if((X == Y) || (X != Y)){

System.out.println("True");

}

else{

System.out.println("False");

}

}

}

**Assignment operators**

Which can be used to assign a value to a variable. Lets suppose variable **A** hold 8 and **B** hold 3.

|  |  |  |
| --- | --- | --- |
| Operator | Example (int A=8, B=3) | Result |
| += | A+=B or A=A+B | 11 |
| -= | A-=3 or A=A+3 | 5 |
| \*= | A\*=7 or A=A\*7 | 56 |
| /= | A/=B or A=A/B | 2 |
| %= | A%=5 or A=A%5 | 3 |
| =a=b | Value of b will be assigned to a |  |

There are following assignment operators supported by Java language:

| **Operator** | **Description** |
| --- | --- |
| = | Simple assignment operator, Assigns values from right side operands to left side operand |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand |
| <<= | Left shift AND assignment operator |
| >>= | Right shift AND assignment operator |
| &= | Bitwise AND assignment operator |
| ^= | Bitwise exclusive OR and assignment operator |
| |= | Bitwise inclusive OR and assignment operator |

Example:

public class Main{

public static void main(String args[]){

int X = 60;

int Y = 13;

X += 1;

System.out.println("X+=1 : "+X);

Y<<=1;

System.out.println("Y<<=1 : "+Y);

/\* Return 26 : 13(binary - 00001101) shift one bit left means 26(00011010) \*/

}

}

**Bitwise Operators**

Java defines several bitwise operators like &, | etc which can be applied to the integer types(long, int, short, char, and byte).

Bitwise operator works on bits(0 or 1) and perform bit by bit operation. Assume if x = 60; and y = 13; Now in binary format they will be as follows:

x = 0011 1100

y = 0000 1101

-----------------

x&y = 0000 1100

x|y =   0011 1101

x^y =  0011 0001

~x =    1100 0011

The following table lists the bitwise operators:

Assume integer variable X=60 and variable Y=13 then:

| **Operator** | **Description** |
| --- | --- |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. |
| | | Binary OR Operator copies a bit if it exists in eather operand. |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. |

**Ternary operator**

If any operator is used on three operands or variable is known as ternary operator. It can be represented with " ?:

The ternary operator "?:" earns its name because it's the only operator to take three [operands](http://java.about.com/od/o/g/operand.htm). It is a [conditional operator](http://java.about.com/od/c/g/conditionaloperator.htm) that provides a shorter syntax for the if..then..else statement. The first operand is a [boolean expression](http://ruby.about.com/od/control/a/Boolean-Expressions.htm); if the expression is true then the value of the second operand is returned otherwise the value of the third operand is returned:

**boolean expression ? value1 : value2**

Java ternary operator is the only conditional operator that takes three operands. Java ternary operator is a one liner replacement for if-then-else statement and used a lot in java programming. We can use ternary operator to replace switch also as shown in below example.

The first operand in java ternary operator should be a boolean or a statement with boolean result. If the first operand is **true** then java ternary operator returns second operand else it returns third operand.

Syntax of java ternary operator is

result = testStatement ? value1 : value2;

If testStatement is true then value1 is assigned to result variable else value2 is assigned to result variable.

Let’s see java ternary operator example in a simple java program.

package com.journaldev.util;

public class TernaryOperator

{

public static void main(String[] args)

{ System.out.println(getMinValue(4,10));

System.out.println(getAbsoluteValue(-10));

System.out.println(invertBoolean(true));

String str = "Australia";

String data = str.contains("A") ? "Str contains 'A'" : "Str doesn't contains 'A'";

System.out.println(data);

int i = 10;

switch (i)

{

case 5:

System.out.println("i=5");

break;

case 10:

System.out.println("i=10");

break;

default:

System.out.println("i is not equal to 5 or 10");

}

System.out.println((i==5) ? "i=5":((i==10) ? "i=10":"i is not equal to 5 or 10"));

}

private static boolean invertBoolean(boolean b) {

return b ? false:true;

}

private static int getAbsoluteValue(int i) {

return i<0 ? -i:i;

}

private static int getMinValue(int i, int j) {

return (i<j) ? i : j;

}

}

Output of the above ternary operator java program is:

4

10

false

Str contains 'A'

i=10

i=10

***5-Java - Modifier Types***

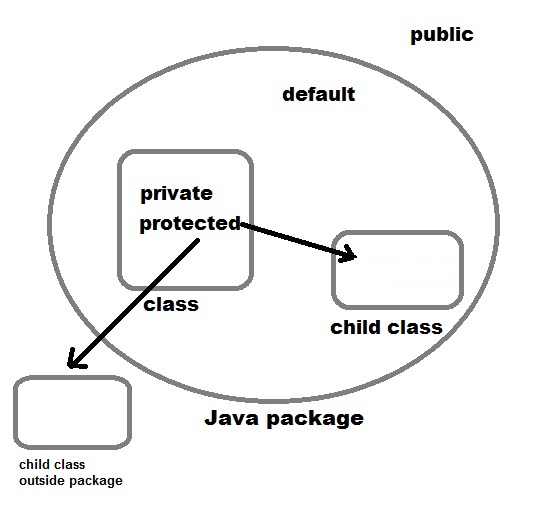
Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

* Java Access Modifiers
* Non Access Modifiers

## Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

* **Default :** Default has scope only inside the same package
* **Public :** Public scope is visible everywhere
* **Protected :** Protected has scope within the package and all sub classes
* **Private :** Private has scope only within the classes



**private**

If a method or variable is marked as private, then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.

If a class is marked as private then no external class an access the class. This doesn't really make so much sense for classes though. Therefore, the access modifier private is mostly used for fields, constructors and methods.

Example :

public class Clock {

private long time = 0;

}

Mostly private access modifier use for fields and make getter, setter method to access these fields.

**default**

The default access level is declared by not writing any access modifier at all. Default access levels means that code inside the class itself + code inside classes in the same package as this class, can access the class, field, constructor or method. Therefore, the default access modifier is also sometimes called a package access modifier.

Subclasses cannot access methods and member variables in the superclass, if they have default accessibility declared, unless the subclass is located in the same package as the superclass.

Example:

public class Clock {

long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

**protected**

The protected acces modifier does the same as the default access, except subclasses can also access protected methods and member variables of the superclass. This is true even if the subclass is not located in the same package as the superclass.

Example:

public class Clock {

protected long time = 0; // time in milliseconds

}

public class SmartClock() extends Clock {

public long getTimeInSeconds() {

return this.time / 1000;

}

}

**public**

The public access modifier means that all code can access the class, field, constructor or method, regardless of where the accessing code is located.

Example:

public class Clock {

public long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

## Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

**The static Modifier:**

**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 variables are also known as class variables. Local variables cannot be declared static.

**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.

Example:

The static modifier is used to create class methods and variables, as in the following example:

public class InstanceCounter {

private static int numInstances = 0;

protected static int getCount() {

return numInstances;

}

private static void addInstance() {

numInstances++;

}

InstanceCounter() {

InstanceCounter.addInstance();

}

public static void main(String[] arguments) {

System.out.println("Starting with " +

InstanceCounter.getCount() + " instances");

for (int i = 0; i < 500; ++i){

new InstanceCounter();

}

System.out.println("Created " +

InstanceCounter.getCount() + " instances");

}

}

This would produce following result:

Started with 0 instances

Created 500 instances

**The final Modifier:**

**final Variables:**

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However the data within the object can be changed. So the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example:

public class Test{

final int value = 10;

// The following are examples of declaring constants:

public static final int BOXWIDTH = 6;

static final String TITLE = "Manager";

public void changeValue(){

value = 12; //will give an error

}

}

**final Methods:**

A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example:

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

public class Test{

public final void changeName(){

// body of method

}

}

**final Classes:**

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

Example:

public final class Test {

// body of class

}

**The abstract Modifier:**

**abstract Class:**

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final. (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example:

abstract class Caravan{

private double price;

private String model;

private String year;

public abstract void goFast(); //an abstract method

public abstract void changeColor();

}

**abstract Methods:**

An abstract method is a method declared with out any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.

If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

Example:

public abstract class SuperClass{

abstract void m(); //abstract method

}

class SubClass extends SuperClass{

// implements the abstract method

void m(){

.........

}

}

**The synchronized Modifier:**

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

Example:

public synchronized void showDetails(){

.......

}

**The transient Modifier:**

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Example:

public transient int limit = 55; // will not persist

public int b; // will persist

**The volatile Modifier:**

The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables which are of type object or private. A volatile object reference can be null.

Example:

public class MyRunnable implements Runnable

{

private volatile boolean active;

public void run()

{

active = true;

while (active) // line 1

{

// some code here

}

}

public void stop()

{

active = false; // line 2

}

}

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1 the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

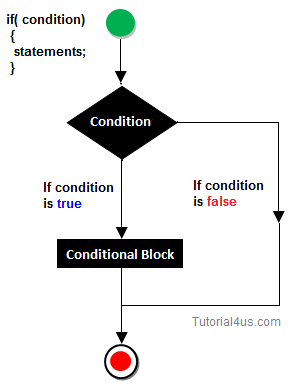
***6-Decision Making Statement***

**Decision making statement** statements is also called selection statement. That is depending on the condition block need to be executed or not which is decided by condition. If the condition is "true" statement block will be executed, if condition is "false" then statement block will not be executed. In java there are three types of decision making statement.

* if
* if-else
* switch

**if-then Statement**

if-then is most basic statement of Decision making statement. It tells to program to execute a certain part of code only if particular condition is true.



**Syntax**

if(condition)

{

Statement(s)

}

Example if statement

**class** Hello

{

**int** a=10;

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

{

**if**(a<15)

{

System.**out**.println("Hello good morning!");

}

}

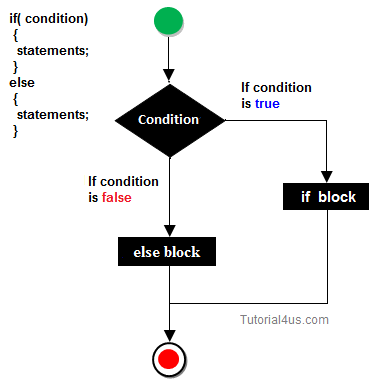
}

**Output**

Hello good morning

**if-else statement**

In general it can be used to execute one block of statement among two blocks, in java language **if** and **else** are the keyword in java.



Syntax

if(condition)

{

Statement(s)

}

else

{

Statement(s)

}

In the above syntax whenever condition is true all the if block statement are executed, remaining statement of the program by neglecting. If the condition is false else block statement executed and neglecting if block statements.

**Example if else**

**import** java.util.Scanner;

**class** Oddeven

{

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

{

**int** **no**;

Scanner s=**new** Scanner(System.**in**);

System.**out**.println("Enter any number :");

**no**=s.nextInt();

**if**(**no**%2==0)

{

System.**out**.println("Even number");

}

**else**

{

System.**out**.println("Odd number");

}

}

}

Output

Enter any number :

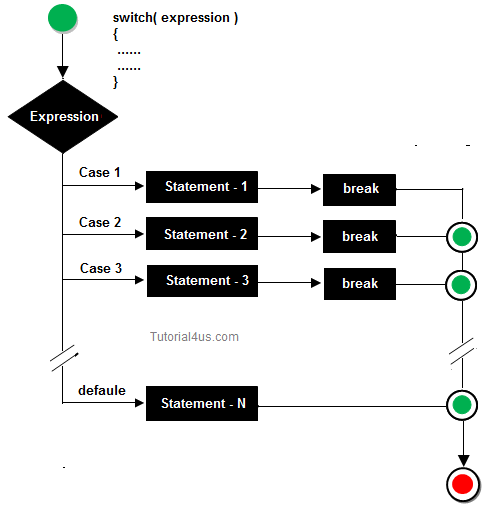
10

Even number

**Switch Statement**

The **switch** statement in java language is used to execute the code from multiple conditions or case. It is same like if else-if ladder statement.

A switch statement work with byte, short, char and int primitive data type, it also works with enumerated types and string.



**Syntax**

switch(expression/variable)

{

case value:

//statements

// any number of case statements

break; //optional

default: //optional

//statements

}

**Rules for apply switch statement**

With switch statement use only byte, short, int, char data type (float data type is not allowed). You can use any number of case statements within a switch. Value for a case must be same as the variable in switch.

**Limitations of switch statement**

Logical operators cannot be used with switch statement. For instance

**Example**

**case** k>=20: // not allowed

Example of switch case

**import** java.util.\*;

**class** switchCase

{

**public** **static** **void** main(String arg[])

{

**int** ch;

System.**out**.println("Enter any number (1 to 7) :");

Scanner s=**new** Scanner(System.**in**);

ch=s.nextInt();

**switch**(ch)

{

**case** 1:

System.**out**.println("Today is Monday");

**break**;

**case** 2:

System.**out**.println("Today is Tuesday");

**break**;

**case** 3:

System.**out**.println("Today is Wednesday");

**break**;

**case** 4:

System.**out**.println("Today is Thursday");

**break**;

**case** 5:

System.**out**.println("Today is Friday");

**break**;

**case** 6:

System.**out**.println("Today is Saturday");

**break**;

**case** 7:

System.**out**.println("Today is Sunday");

**default**:

System.**out**.println("Only enter value 1 to 7");

}

}

}

Output

Enter any number (1 to 7):

5

Today is Friday

***7-Java - Loop control***

Loop is very common control flow statement in programming languages such as java. We are going to describe the basics of “java loop”. In this post, we will learn various ways to use loop in day-to-day programming habits.

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

There are four types of loops:

1. For loop

2. For each loop

3. While loop

4. Do..While loop

**For Loop**

It is structured around a finite set of repetitions of code. So if you have a particular block of code that you want to have run over and over again a specific number of times the For Loop is helpful.

Syntax:

for(initilization; conditional expression; increment expression)

{

//repetition code here

}

Example:

public class Example {

public static void main(String args[]) {

for(int x = 50; x < 55; x++) {

System.out.println("Value of x : " + x );

}

}

}

Output:

Value of x : 50;

Value of x : 51;

Value of x : 52;

Value of x : 53;

Value of x : 54;

**For each Loop**

This loop is supported from Java 5. For each loop is mainly used for iterate the Array, List etc.

Syntax:

for(declaration : expression)

{

//Code Here

}

Example:

public class Example {

public static void main(String args[]){

List list = new ArrayList();

list.add(10);

list.add(20);

list.add(30);

System.out.print("List Value = ");

for(int x : list){

System.out.print( x );

System.out.print(",");

}

String [] names ={"abc", "xyz", "test", "example"};

System.out.println("String Array value = ");

for( String name : names ) {

System.out.print( name );

System.out.print(",");

}

}

}

Output:

List Value = 10,20,30

String Array value = abc,xyz,test,example

**While Loop**

Another looping strategy is known as the While Loop. The While Loop is good when you don’t want to repeat your code a**specific** number of times, rather, you want to **keep** **looping** through your code until a **certain** **condition** **is** **met**.

Syntax:

while(Boolean\_expression)

{

//Repetition Code Here

}

Example:

public class Example {

public static void main(String args[]) {

int x = 50;

while( x < 55 ) {

System.out.println("Value of x : " + x );

x++;

}

}

}

Output:

Value of x : 50

Value of x : 51

Value of x : 52

Value of x : 53

Value of x : 54

**Do..While Loop**

This type of loop is used in very rare cases because it does the same thing as a while loop does, except that a do..while loop is guaranteed to execute at least on time.

Syntax:

do

{

//Code Here

}while(Boolean\_expression);

Example:

public class Test {

public static void main(String args[]){

int x = 50;

do{

System.out.println("Value of x : " + x );

x++;

}while( x < 50 );

}

}

Output:

Value of x : 50

Above example first execute code inside loop and then check condition that’s why it’s display 50.

***8-Java - String***

A string in literal terms is a sequence of characters like the word “hello”. Hey, did you say characters, isn’t it a primitive data type in Java. Yes, so in technical terms, the basic Java String is basically an array of characters. In the java programming language, string is object.

**Immutable String**

Java String is a immutable object. For an immutable object you cannot modify any of its attribute’s values. Once you have created a java String object it cannot be modified to some other object or a different String. A reference to a java String instance is mutable. There are multiple ways to make an object immutable. Simple and straight forward way is to make all the attributes of that class as final. Java String has all attributes marked as final except hash field.

We all know java String is immutable but do we know why java String is immutable? Main reason behind it is for better performance. Creating a copy of existing java String is easier as there is no need to create a new instance but can be easily created by pointing to already existing String. This saves valuable primary memory.

String Syntax:

String as an array of characters like:

char[] charArray = {'V','I','S','I','O','N'};

String str = new String(charArray);

String in Java as:

String srt = new String("VISION");

**Initialization of String:**

JVM maintains a memory pool for String. When you create a String, first this memory pool is scanned. If the instance already exists then this new instance is mapped to the already existing instance. If not, a new java String instance is created in the memory pool.

String str1 = "VISION";

String srt2 = new String();

String str3 = new String("VISION");

String str4 = new String(char[]);

String str5 = new String(byte[]);

String str6 = new String(new StringBuffer());

String str7 = new String(new StringBuilder());

We have an empty constructor for String. It is odd, java String is immutable and you have an empty constructur which does nothing but create a empty String. I don’t see any use for this constructor, because after you create a String you cannot modify it.

**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.

Example:

public class Main{

public static void main(String args[]){

String str = new String("visions");

System.out.println(str);

}

}

Output:

visions

**String length:**

String length() method returns number of characters contained in the String object.

Example:

public class Main{

public static void main(String args[]){

String str = new String("visions");

System.out.println("String length : "+str.length());

}

}

Output:

String length : 7

**Concatenating String:**

Concatenating is joining of two or more string into one string.

We have two string str1=”visions” and str2=”developer”. If we add these two strings, we should be having a result as str3  =”visionsdeveloper”. String having two methods for concatenating string first is “+”(plus operators) and concat() method.

Example:

public class Main{

public static void main(String args[]){

String str1 = "visions";

String str2 = "developer";

String srt3 = str1+str2;

System.out.println(str3);

String str4 = str1.concat(str2);

System.out.println(str4);

}

}

Output:

visionsdeveloper

visionsdeveloper

**Note:** Don’t use “+” operator because it is not good for performance.

# Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](http://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](http://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](http://www.javatpoint.com/java-string-format) | returns formatted string |
| 4 | [static String format(Locale l, String format, Object... args)](http://www.javatpoint.com/java-string-format) | returns formatted string with given locale |
| 5 | [String substring(int beginIndex)](http://www.javatpoint.com/java-string-substring) | returns substring for given begin index |
| 6 | [String substring(int beginIndex, int endIndex)](http://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index |
| 7 | [boolean contains(CharSequence s)](http://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](http://www.javatpoint.com/java-string-join) | returns a joined string |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](http://www.javatpoint.com/java-string-join) | returns a joined string |
| 10 | [boolean equals(Object another)](http://www.javatpoint.com/java-string-equals) | checks the equality of string with object |
| 11 | [boolean isEmpty()](http://www.javatpoint.com/java-string-isempty) | checks if string is empty |
| 12 | [String concat(String str)](http://www.javatpoint.com/java-string-concat) | concatinates specified string |
| 13 | [String replace(char old, char new)](http://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value |
| 14 | [String replace(CharSequence old, CharSequence new)](http://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified CharSequence |
| 15 | [static String equalsIgnoreCase(String another)](http://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](http://www.javatpoint.com/java-string-split) | returns splitted string matching regex |
| 17 | [String[] split(String regex, int limit)](http://www.javatpoint.com/java-string-split) | returns splitted string matching regex and limit |
| 18 | [String intern()](http://www.javatpoint.com/java-string-intern) | returns interned string |
| 19 | [int indexOf(int ch)](http://www.javatpoint.com/java-string-indexof) | returns specified char value index |
| 20 | [int indexOf(int ch, int fromIndex)](http://www.javatpoint.com/java-string-indexof) | returns specified char value index starting with given index |
| 21 | [int indexOf(String substring)](http://www.javatpoint.com/java-string-indexof) | returns specified substring index |
| 22 | [int indexOf(String substring, int fromIndex)](http://www.javatpoint.com/java-string-indexof) | returns specified substring index starting with given index |
| 23 | [String toLowerCase()](http://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase. |
| 24 | [String toLowerCase(Locale l)](http://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase using specified locale. |
| 25 | [String toUpperCase()](http://www.javatpoint.com/java-string-touppercase) | returns string in uppercase. |
| 26 | [String toUpperCase(Locale l)](http://www.javatpoint.com/java-string-touppercase) | returns string in uppercase using specified locale. |
| 27 | [String trim()](http://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](http://www.javatpoint.com/java-string-valueof) | converts given type into string. It is overloaded. |

**Java StringBuffer class**

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

**Important Constructors of StringBuffer class**

StringBuffer(): creates an empty string buffer with the initial capacity of 16.

StringBuffer(String str): creates a string buffer with the specified string.

StringBuffer(int capacity): creates an empty string buffer with the specified capacity as length.

**Important methods of StringBuffer class**

1. public synchronized StringBuffer append(String s): is 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.
2. public synchronized StringBuffer insert(int offset, String s): is 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.
3. public synchronized StringBuffer replace(int startIndex, int endIndex, String str): is used to replace the string from specified startIndex and endIndex.
4. public synchronized StringBuffer delete(int startIndex, int endIndex): is used to delete the string from specified startIndex and endIndex.
5. public synchronized StringBuffer reverse(): is used to reverse the string.
6. public int capacity(): is used to return the current capacity.
7. public void ensureCapacity(int minimumCapacity): is used to ensure the capacity at least equal to the given minimum.
8. public char charAt(int index): is used to return the character at the specified position.
9. public int length(): is used to return the length of the string i.e. total number of characters.
10. public String substring(int beginIndex): is used to return the substring from the specified beginIndex.
11. public String substring(int beginIndex, int endIndex): is used to return the substring from the specified beginIndex and endIndex.

**What is mutable string**

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

**StringBuffer reverse() method**

The reverse() method of StringBuilder class reverses the current string.

class A

{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

**Java StringBuilder class**

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

**Important Constructors of StringBuilder class**

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.

**Important methods of StringBuilder class**

1. **public StringBuilder append(String s)**-is 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.
2. **public StringBuilder insert(int offset, String s)**- is 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.
3. **public StringBuilder replace(int startIndex, int endIndex, String str)**- is used to replace the string from specified startIndex and endIndex.
4. **public StringBuilder delete(int startIndex, int endIndex)**- is used to delete the string from specified startIndex and endIndex.
5. **public StringBuilder reverse()**- is used to reverse the string.
6. **public int capacity()-** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity)-** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index)**- is used to return the character at the specified position.
9. **public int length()**- is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex)** -is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex)-**is used to return the substring from the specified beginIndex and endIn

**Example-StringBuilder ensureCapacity() method**

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

class A{

public static void main(String args[]){

StringBuilder sb=new StringBuilder();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

}

**Difference between String and StringBuffer**

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you cancat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

**Difference between StringBuffer and StringBuilder**

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

***9-Java - Arrays***

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

**The first element of array start with zero.**

Using an array in your program is a 3 step process:

1. Declaring you array.
2. Constructing your array.
3. Initializing your array.

**Declaring Array:-**

Syntax:

elementType[] arrayName;

Or

elementType arrayName[];

Example:

int[] intArray;

int intArray[];

**Constructing Array:-**

Syntax:

new elementType[size];

Example:

int[] intArray = new int[10]; // Defines that intArray will store 10 integer values

int intArray[] = new int[10];

**Initializing Array:-**

Syntax:

arrayName[element 0,1,2?.. N] = value;

Example:

intArray[0] = 10; // Assign an integer value 10 to the first element 0 of the array

intArray[1] = 20;

**Declaring and Initializing Array:-**

Syntax:

elementType[] arrayName = {values1,values2,? valueN};

Example:

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

Array Example:

public class Main {

public static void main(String[] args) {

String[] names = new String[3];

names[0] = "A";

names[1] = "ABC";

names[2] = "XYZ";

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

System.out.println(names[i]);

}

//this line should throw an exception

//System.out.println(names[6]);

}

}

## ****Array are passed by reference:****

Arrays are passed to functions by reference, or as a pointer to the original. This means anything you do to the Array inside the function affects the original.

Example:

public class Main{

public static void passByRefrence(String a[]){

a[0] = "Z";

}

public static void main(String args[]){

String[] b = {"A","B","C"};

System.out.println("Before Function call : "+b[0]);

Main.passByRefrence(b);

System.out.println("After Function call : "+b[0]);

}

}

Output:

Before Function call : A

After Function call : Z

## Multidimensional Arrays:

Multidimensional arrays, are arrays of arrays.

Syntax:

elementType[][] arrayName = new elementType[size][size];

Example:

int[][] intArrays = new int[4][5];

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.

You can allocate the remaining dimensions separately.

In Java the length of each array in a multidimensional array is under your control.

Example:

int multi[][] = new int[2][];

multi[0] = new int[5];

multi[1] = new int[4];

## Array of Objects:

It is possible to create array of objects of user created class.

Example:

class Employee{

int id;

String name;

public void setData(int id, String name){

this.id = id;

this.name = name;

}

public void displayData()

{

System.out.println("Employee ID : "+this.id);

System.out.println("Employee Name : "+this.name);

}

}

class Main

{

public static void main(String args[])

{

Employee[] emp = new Employee[2];

emp[0].setData(1,"ABC");

emp[1].setData(2,"XYZ");

emp[0].displayData();

emp[1].displayData(); }

}

Output:

Employee ID : 1

Employee Name : ABC

Employee ID : 2

Employee Name : XYZ

**1-Multi-Scenario Array-**

public class TestArray {

public static void main(String[] args) {

double[] myList = {1.9, 2.9, 3.4, 3.5};

// Print all the array elements

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

System.out.println(myList[i] + " ");

}

// Summing all elements

double total = 0;

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

total += myList[i];

}

System.out.println("Total is " + total);

// Finding the largest element

double max = myList[0];

for (int i = 1; i < myList.length; i++) {

if (myList[i] > max) max = myList[i];

}

System.out.println("Max is " + max);

}

}

**2-Multi-Scenario Array-**

public class ArrayExamples

{ public static void main(String[] args)

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

findAndPrintPairs(list, 5);

bubblesort(list);

showList(list);

list = new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

bubblesort(list);

showList(list);

list = new int[]{11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2};

bubblesort(list);

showList(list);

list = new int[]{1};

bubblesort(list);

showList(list);

}

// pre: list != null, list.length > 0

// post: return index of minimum element of array

public static int findMin(int[] list)

{ assert list != null && list.length > 0 : "failed precondition";

int indexOfMin = 0;

for(int i = 1; i < list.length; i++)

{ if(list[i] < list[indexOfMin])

{ indexOfMin = i;

}

}

return indexOfMin;

}

/\*

\*pre: list != null, newSize >= 0

\*post: nothing. the method does not succeed it resizing the

\* argument

\*/

public static void badResize(int[] list, int newSize)

{ assert list != null && newSize >= 0 : "failed precondition";

int[] temp = new int[newSize];

int limit = Math.min(list.length, newSize);

for(int i = 0; i < limit; i++)

{ temp[i] = list[i];

}

// uh oh!! Changing pointer, not pointee. This breaks the

// relationship between the parameter and argument

list = temp;

}

/\*

\*pre: list != null, newSize >= 0

\*post: returns an array of size newSize. Elements from 0 to newSize - 1

\* will be copied into the new array

\*/

public static int[] goodResize(int[] list, int newSize)

{ assert list != null && newSize >= 0 : "failed precondition";

int[] result = new int[newSize];

int limit = Math.min(list.length, newSize);

for(int i = 0; i < limit; i++)

{ result[i] = list[i];

}

return result;

}

/\*

\*pre: list != null

\*post: prints out the indices and values of all pairs of numbers

\*in list such that list[a] + list[b] = target

\*/

public static void findAndPrintPairs(int[] list, int target)

{ assert list != null : "failed precondition";

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

{ for(int j = i + 1; j < list.length; j++)

{ if(list[i] + list[j] == target)

{ System.out.println("The two elements at indices " + i + " and " + j

+ " are " + list[i] + " and " + list[j] + " add up to " + target);

}

}

}

}

/\*

\*pre: list != null;

\*post: sort the elements of list so that they are in ascending order

\*/

public static void bubblesort(int[] list)

{

assert list != null : "failed precondition";

int temp;

boolean changed = true;

for(int i = 0; i < list.length && changed; i++)

{ changed = false;

for(int j = 0; j < list.length - i - 1; j++)

{ assert (j > 0) && (j + 1 < list.length) : "loop counter j " + j +

"is out of bounds.";

if(list[j] > list[j+1])

{ changed = true;

temp = list[j + 1];

list[j + 1] = list[j];

list[j] = temp;

}

}

}

assert isAscending( list );

}

public static void showList(int[] list)

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

System.out.print( list[i] + " " );

System.out.println();

}

/\* pre: list != null

post: return true if list is sorted in ascedning order, false otherwise

\*/

public static boolean isAscending( int[] list )

{ boolean ascending = true;

int index = 1;

while( ascending && index < list.length )

{ assert index >= 0 && index < list.length;

ascending = (list[index - 1] <= list[index]);

index++;

}

return ascending;

}

}

**2-Array sort and search:-**

import java.util.Arrays;

public class MainClass

{

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

int array[] = { 2, 5, -2, 6, -3, 8, 0, -7, -9, 4 };

Arrays.sort(array);

printArray("Sorted array", array);

int index = Arrays.binarySearch(array, 2);

System.out.println("Found 2 @ " + index);

}

private static void printArray(String message, int array[]) {

System.out.println(message

+ ": [length: " + array.length + "]");

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

if(i != 0){

System.out.print(", ");

}

System.out.print(array[i]);

}

System.out.println();

}

}

**2-Finding the longest string in a Java string array**

public class JavaLongestStringInStringArray {

public static String getLongestString(String[] array) {

int maxLength = 0;

String longestString = null;

for (String s : array) {

if (s.length() > maxLength) {

maxLength = s.length();

longestString = s;

}

}

return longestString;

}

public static void main(String[] args) {

String[] toppings = {"Cheese", "Pepperoni", "Black Olives"};

String longestString = getLongestString(toppings);

System.out.format("longest string: '%s'\n", longestString);

}

}

**3-Java program that loops over array in reverse**

public class Program {

public static void main(String[] args) {

boolean[] values = { false, true, true, true };

// Loop over array elements in reverse order.

for (int i = values.length - 1; i >= 0; i--) {

System.out.println(values[i]);

}

}

}

**4-Java that merges two arrays**

import java.util.Arrays;

public class Program {

public static void main(String[] args) {

int[] values = { 10, 20, 30 };

int[] values2 = { 100, 200, 300 };

// Merge the two arrays with for-loops.

int[] merge = new int[values.length + values2.length];

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

merge[i] = values[i];

}

for (int i = 0; i < values2.length; i++) {

merge[i + values.length] = values2[i];

}

// Display the merged array.

System.out.println(Arrays.toString(merge));

}

}

**5-How to Remove Duplicates from Array without using Java Collection API-**

import java.util.Arrays;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class TechnicalInterviewTest

{

private static final Logger logger = LoggerFactory.getLogger(TechnicalInterviewTest.class);

public static void main(String args[])

{

int[][] test = new int[][]{

{1, 1, 2, 2, 3, 4, 5},

{1, 1, 1, 1, 1, 1, 1},

{1, 2, 3, 4, 5, 6, 7},

{1, 2, 1, 1, 1, 1, 1},};

for (int[] input : test) {

System.out.println("Array with Duplicates : " + Arrays.toString(input));

System.out.println("After removing duplicates : " + Arrays.toString(removeDuplicates(input)));

}

}

/\*

\* Method to remove duplicates from array in Java, without using

\* Collection classes e.g. Set or ArrayList. Algorithm for this

\* method is simple, it first sort the array and then compare adjacent

\* objects, leaving out duplicates, which is already in the result.

\*/

public static int[] removeDuplicates(int[] numbersWithDuplicates) {

// Sorting array to bring duplicates together

Arrays.sort(numbersWithDuplicates);

int[] result = new int[numbersWithDuplicates.length];

int previous = numbersWithDuplicates[0];

result[0] = previous;

for (int i = 1; i < numbersWithDuplicates.length; i++) {

int ch = numbersWithDuplicates[i];

if (previous != ch) {

result[i] = ch;

}

previous = ch;

}

return result;

}

}

Output:-

Array with Duplicates : [1, 1, 2, 2, 3, 4, 5]

After removing duplicates : [1, 0, 2, 0, 3, 4, 5]

Array with Duplicates : [1, 1, 1, 1, 1, 1, 1]

After removing duplicates : [1, 0, 0, 0, 0, 0, 0]

Array with Duplicates : [1, 2, 3, 4, 5, 6, 7]

After removing duplicates : [1, 2, 3, 4, 5, 6, 7]

Array with Duplicates : [1, 2, 1, 1, 1, 1, 1]

After removing duplicates : [1, 0, 0, 0, 0, 0, 2]

***10-Java - Exception Handling***

**What is Error?**

Error are exceptional scenario that are out of scope for application and it’s not possible to recover. E.g Hardware Failure, Network failure.

**What is an exception?**

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn’t continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled. We will cover the handling part later in this same tutorial.

**When an exception can occur?**

Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

**Reasons for Exceptions**

There can be several reasons for an exception. For example, following situations can cause an exception – Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on.

**Difference between error and exception**

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

**Advantages of Exception Handling**

1. Exception handling allows us to control the normal flow of the program by using exception handling in program.
2. It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
3. It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.
4. Why to handle exception?
5. If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

**printStackTrace()** – It helps the programmer to understand where the actual problem occurred. printStacktrace() is a method of the class Throwable of java.lang package. It prints several lines in the output console. The first line consists of several strings. It contains the name of the Throwable sub-class & the package information. From second line onwards, it describes the error position/line number beginning with "at".

The last line always describes the destination affected by the error/exception. The second last line informs us about the next line in the stack where the control goes after getting transfer from the line number described in the last line. The errors/exceptions represents the output in the form a stack, which were fed into the stack by fillInStackTrace() method of Throwable class, which itself fills in the program control transfer details into the execution stack. The lines starting with at, are nothing but the values of the execution stack. In this way the programmer can understand where in code the actual problem is.

Along with the printStackTrace() method, it's a good idea to use e.getmessage().

**Keywords use in Exception Handling**

**throw:-**It is possible for your program to throw an exception explicitly, using the throw statement.

throw ThrowableInstance;

ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

The flow of execution stop immediately after throw statement, any statement not executed after this statement within same try ..catch block.

**throws:-**If method is capable of causing an exception that it does not handle, it must specify this exception to handle by its caller method. This throws clause use with method declaration.

**try-catch:-**We use try-catch block for exception handling in our application.try is the start of the block and catch is the end of block in exception handling. We can have multiple catch with try and try-catch block nested also. catch block required parameter of type Exception.

**finally:** finally block is optional and can we use only with try-catch. finally block always executed whether exception occur or not. finally block always use for close the resource.

**Types of exceptions**

There are two types of exceptions

1)Checked exceptions

2)Unchecked exceptions

**Checked exceptions**

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, it will give compilation error.

**Examples of Checked Exceptions :-**

* ClassNotFoundException
* IllegalAccessException
* NoSuchFieldException
* EOFException etc.
* SQLException
* IOException
* DataAccessException
* ClassNotFoundException
* InvocationTargetException

**Checked and unchecked exceptions in java with examples**

What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error. It is named as checked exception because these exceptions are checked at Compile time.

Lets understand this with this example: In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where an checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

/\*This constructor FileInputStream(File filename)

\* throws FileNotFoundException which is a checked

\* exception\*/

fis = new FileInputStream("B:/myfile.txt");

int k;

/\*Method read() of FileInputStream class also throws

\* a checked exception: IOException\*/

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

/\*The method close() closes the file input stream

\* It throws IOException\*/

fis.close();

}

}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.

**How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.**

**Method 1: Declare the exception using throws keyword.**

As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. Th reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare that too like this public static void main(String args[]) throws IOException, FileNotFoundException.

import java.io.\*;

class Example {

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

{

FileInputStream fis = null;

fis = new FileInputStream("B:/myfile.txt");

int k;

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}

}

**Output:**

File content is displayed on the screen.

**Method 2: Handle them using try-catch blocks.**

The above approach is not good at all. It is not a best exception handling practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

try{

fis = new FileInputStream("B:/myfile.txt");

}catch(FileNotFoundException fnfe){

System.out.println("The specified file is not " +

"present at the given path");

}

int k;

try{

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}catch(IOException ioe){

System.out.println("I/O error occurred: "+ioe);

}

}

}

This code will run fine and will display the file content.

**What are Unchecked exceptions?**

**Unchecked Exceptions**

These exceptions need not be included in any method’s throws list because compiler does not check to see if a method handles or throws these exceptions.

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of RuntimeException class.

Lets understand this with an example:

class Example {

public static void main(String args[])

{

int num1=10;

int num2=0;

/\*Since I'm dividing an integer with 0

\* it should throw ArithmeticException\*/

int res=num1/num2;

System.out.println(res);}}

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they are being checked at runtime. Lets see another example.

class Example {

public static void main(String args[])

{

int arr[] ={1,2,3,4,5};

/\*My array has only 5 elements but

\* I'm trying to display the value of

\* 8th element. It should throw

\* ArrayIndexOutOfBoundsException\*/

System.out.println(arr[7]);

}

}

This code would also compile successfully since ArrayIndexOutOfBoundsException is also an unchecked exception.

Note: It doesn’t mean that compiler is not checking these exceptions so we shouldn’t handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn’t exist in array so that user would be able to correct the issue.

class Example {

public static void main(String args[])

{

try{

int arr[] ={1,2,3,4,5};

System.out.println(arr[7]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("The specified index does not exist " +

"in array. Please correct the error.");

}

}

}

Here are the few most frequently seen unchecked exceptions –

**Examples of Unchecked Exceptions:-**

* ArithmeticException
* ArrayIndexOutOfBoundsException
* NullPointerException
* NegativeArraySizeException etc.
* IllegalArgumentException

**Try Catch in Java – Exception handling**

**What is Try Block?**

The try block contains a block of program statements within which an exception might occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must followed by a Catch block or Finally block or both.

**What is Catch Block?**

A catch block must be associated with a try block. The corresponding catch block executes if an exception of a particular type occurs within the try block. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

**Flow of try catch block**

If an exception occurs in try block then the control of execution is passed to the catch block from try block. The exception is caught up by the corresponding catch block. A single try block can have multiple catch statements associated with it, but each catch block can be defined for only one exception class. The program can also contain nested try-catch-finally blocks.

After the execution of all the try blocks, the code inside the finally block executes. It is not mandatory to include a finally block at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch blocks.

**An example of Try catch in Java**

class Example1 {

public static void main(String args[]) {

int num1, num2;

try {

// Try block to handle code that may cause exception

num1 = 0;

num2 = 62 / num1;

System.out.println("Try block message");

} catch (ArithmeticException e) {

// This block is to catch divide-by-zero error

System.out.println("Error: Don't divide a number by zero");

}

System.out.println("I'm out of try-catch block in Java.");

}

}

**Output:**

Error: Don't divide a number by zero

I'm out of try-catch block in Java.

**Multiple catch blocks in Java**

1. A try block can have any number of catch blocks.

2. A catch block that is written for catching the class Exception can catch all other exceptions

1. A try block can have any number of catch blocks.

2. A catch block that is written for catching the class Exception can catch all other exceptions

3. If multiple catch blocks are present in a program then the above mentioned catch block should be placed at the last as per the exception handling best practices.

4. If the try block is not throwing any exception, the catch block will be completely ignored and the program continues.

5. If the try block throws an exception, the appropriate catch block (if one exists) will catch it

–catch(ArithmeticException e) is a catch block that can catch ArithmeticException

–catch(NullPointerException e) is a catch block that can catch NullPointerException

6. All the statements in the catch block will be executed and then the program continues.

***Example of Multiple catch blocks***

class Example2

{

public static void main(String args[])

{

try{

int a[]=new int[7];

a[4]=30/0;

System.out.println("First print statement in try block");

}

catch(ArithmeticException e){

System.out.println("Warning: ArithmeticException");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Warning: ArrayIndexOutOfBoundsException");

}

catch(Exception e){

System.out.println("Warning: Some Other exception");

}

System.out.println("Out of try-catch block...");

}

}

**Output:**

Warning: ArithmeticException

Out of try-catch block...

**Nested try catch example – explanation**

class Nest{

public static void main(String args[]){

//Parent try block

try{

//Child try block1

try{

System.out.println("Inside block1");

int b =45/0;

System.out.println(b);

}

catch(ArithmeticException e1){

System.out.println("Exception: e1");

}

//Child try block2

try{

System.out.println("Inside block2");

int b =45/0;

System.out.println(b);

}

catch(ArrayIndexOutOfBoundsException e2){

System.out.println("Exception: e2");

}

System.out.println("Just other statement");

}

catch(ArithmeticException e3){

System.out.println("Arithmetic Exception");

System.out.println("Inside parent try catch block");

}

catch(ArrayIndexOutOfBoundsException e4){

System.out.println("ArrayIndexOutOfBoundsException");

System.out.println("Inside parent try catch block");

}

catch(Exception e5){

System.out.println("Exception");

System.out.println("Inside parent try catch block");

}

System.out.println("Next statement..");

}

}

**Output:**

Inside block1

Exception: e1

Inside block2

Arithmetic Exception

Inside parent try catch block

Next statement..

The above example shows Nested try catch use in Java. You can see that there are two try-catch block inside main try block’s body. I’ve marked them as block 1 and block 2 in above example.

Block1: I have divided an integer by zero and it caused an arithmetic exception however the catch of block1 is handling arithmetic exception so "Exception: e1" got printed.

Block2: In block2 also, ArithmeticException occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException so in this case control jump back to Main try-catch(parent) body. Since catch of parent try block is handling this exception that’s why “Inside parent try catch block” got printed as output.

**Use of throws keyword in Java**

**Throw keyword in java-**

Throw keyword is used to throw exception manually.

Whenever it is required to suspend the execution of the functionality based on the user defined logical error condition we will use this throw keyword to throw exception. So we need to handle these exceptions also using try catch blocks.

**Java simple example Program to explain use of throw keyword in java**

package exceptions;

public class ThrowKeyword

{

public static void main(String[] args)

{

try {

throw new ArithmeticException();

} catch (Exception e) {

System.out.println(e);

e.printStackTrace();

}}}

**Output:**

java.lang.ArithmeticException

java.lang.ArithmeticException

at exceptions.ThrowKeyword.main(ThrowKeyword.java:11)

**Rules to use "throw" keyword in java-**

throw keyword must follow Throwable type of object.

It must be used only in method logic.

Since it is a transfer statement , we can not place statements after throw statement. It leads to compile time error Unreachable code

**We can throw user defined exception using throw keyword.-**

package com.exceptions;

public class ThrowDemo {

public boolean isValidForVote(int age){

try{

if(age<18){

throw new InvalidAgeException ("Invalid age for voting");

}

}catch(Exception e){

System.out.println(e);

}

return false;

}

public static void main(String agrs[]){

ThrowDemo obj= new ThrowDemo();

obj.isValidForVote(17);

}}

**Output:**

exceptions.InvalidAgeException: Invalid age for voting

**We can throw predefined exceptions using throw keyword-**

package com.instanceofjava;

public class ThrowKeyword{

public void method(){

try{

throw new NullPointerException("Invalid age for voting");

}

}catch(Exception e){

System.out.println(e);

}

}

public static void main(String agrs[]){

ThrowKeyword obj= new ThrowKeyword();

obj.method();

}

}

**Output:**

java.lang.NullPointerException: Invalid age for voting

**Throws Keyword Example in Java**

Throws Keyword-Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the throws keyword.

**NOTE : It is necessary for all exceptions, except the exceptions of type Error and RuntimeException, or any of their subclass.**

As we know that there are two types of exception – checked and unchecked. Checked exceptions (compile time) are the one which forces the programmer to handle it, without which the program doesn’t compile successfully. While unchecked exception (Runtime) doesn’t get checked during compilation. “Throws keyword” is mainly used for handling checked exception as using throws we can declare multiple exceptions in one go. Let’s understand this with the help of an example.

In this example the method “mymethod” is throwing two checked exceptions so we have declared those exceptions in the method signature using throws Keyword. If we do not declare these exceptions then the program will throw a compilation error:-

**package beginnersbook.com;**

import java.io.\*;

public class ThrowExample

{

void mymethod(int num)throws IOException, ClassNotFoundException{

if(num==1)

throw new IOException("Exception Message1");

else

throw new ClassNotFoundException("Exception Message2");

}

}

class Demo

{

public static void main(String args[]){

try{

ThrowExample obj=new ThrowExample();

obj.mymethod(1);

}

catch(Exception ex)

{

System.out.println(ex);

} }}

**Output:**

java.io.IOException: Exception Message1

Since we passed the argument as 1 during the function call, the program thrown IOException.

**Difference between throw and throws in Java**

|  |  |  |
| --- | --- | --- |
| **No.** | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. |
| public void method()throws IOException,SQLException. |

**Finally clause**

A finally keyword is used to create a block of code that follows a try block. A finally block of code always executes whether or not exception has occurred. Using a finally block, lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

**Finally clause in exception handling in java**

Example demonstrating finally Clause

Class ExceptionTest

{

public static void main(String[] args)

{

int a[]= new int[2];

System.out.println("out of try");

try

{

System.out.println("Access invalid element"+ a[3]);

/\* the above statement will throw ArrayIndexOutOfBoundException \*/

}

finally

{

System.out.println("finally is always executed.");

}

}

}

**Output :**

Out of try

finally is always executed.

Exception in thread main java. Lang. exception array Index out of bound exception.

You can see in above example even if exception is thrown by the program, which is not handled by catch block, still finally block will get executed.

**Below example illustrates finally block execution when exception occurs in try block but doesn’t get handled in catch block.**

class Example2{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/0;

System.out.println(num);

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("ArrayIndexOutOfBoundsException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

**Output:**

First statement of try block

finally block

Exception in thread "main" java.lang.ArithmeticException: / by zero

at beginnersbook.com.Example2.main(Details.java:6)

**Below example illustrates execution of finally, when exception occurs in try block and handled in catch block.**

class Example3{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("ArithmeticException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

**Output:**

First statement of try block

ArithmeticException

finally block

Out of try-catch-finally block

***11- Java – Files(txt,xml,exl) I/O***

**IO Stream**

Java performs I/O through Streams. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Java encapsulates Stream under java.io package. Java defines two types of streams. They are,

**Byte Stream :** It provides a convenient means for handling input and output of byte.

**Character Stream** : It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

**Some important Byte stream classes.**

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 - Abstract class that describe stream input.

OutputStream- Abstract class that describe stream output.

PrintStream - Output Stream that contain print() and println() method

**Some important Charcter stream classes.**

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

OutputStreamReader - Output stream that translate character to byte.

PrintWriter - Output Stream that contain print() and println() method.

Reader - Abstract class that define character stream input

Writer - Abstract class that define character stream output

**Working on Txt/properties files**

**1-Program to take String input from Keyboard in Java**

import java.io.\*;

class MyInput

{

public static void main(String[] args)

{

String text;

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

text = br.readLine(); //Reading String

System.out.println(text);

}

}

2-How to create a file in Java-

The File.createNewFile() method is used to create a file in Java, and return a boolean value : true if the file is created successful; false if the file is already exists or the operation failed.

Exampe-

package com.mkyong.file;

import java.io.File;

import java.io.IOException;

public class CreateFileExample

{

public static void main( String[] args )

{

try {

File file = new File("c:\\newfile.txt");

if (file.createNewFile()){

System.out.println("File is created!");

}else{

System.out.println("File already exists.");

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**3-Program to read from a file using BufferedReader class**

import java. Io \*;

class ReadTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

BufferedReader br = new BufferedReader(new FileReader(fl)) ;

String str;

while ((str=br.readLine())!=null)

{

System.out.println(str);

}

br.close();

fl.close();

}

catch (IOException e)

{ e.printStackTrace(); }

}

}

4-Program to write to a File using FileWriter class

import java. Io \*;

class WriteTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

String str="Write this string to my file";

FileWriter fw = new FileWriter(fl) ;

fw.write(str);

fw.close();

fl.close();

}

catch (IOException e)

{ e.printStackTrace(); }

}

}

**5-How to delete file in Java**

import java.io.File;

public class DeleteFileExample

{

public static void main(String[] args)

{

try{

File file = new File("c:\\logfile20100131.log");

if(file.delete()){

System.out.println(file.getName() + " is deleted!");

}else{

System.out.println("Delete operation is failed.");

}

}catch(Exception e){

e.printStackTrace();

}

}

}

**Working on XML files**

DOM provides many handy classes to create XML file easily. Firstly, you have to create a Document with DocumentBuilder class, define all the XML content – node, attribute with Element class. In last, use Transformer class to output the entire XML content to stream output, typically a File.

**6-How to create xml file-using DOM Parser**

File : WriteXMLFile.java – Java class to create a XML file.

package com.mkyong.core;

import java.io.File;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.ParserConfigurationException;

import javax.xml.transform.Transformer;

import javax.xml.transform.TransformerException;

import javax.xml.transform.TransformerFactory;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Attr;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

public class WriteXMLFile {

public static void main(String argv[]) {

try {

DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();

DocumentBuilder docBuilder = docFactory.newDocumentBuilder();

// root elements

Document doc = docBuilder.newDocument();

Element rootElement = doc.createElement("company");

doc.appendChild(rootElement);

// staff elements

Element staff = doc.createElement("Staff");

rootElement.appendChild(staff);

// set attribute to staff element

Attr attr = doc.createAttribute("id");

attr.setValue("1");

staff.setAttributeNode(attr);

// shorten way

// staff.setAttribute("id", "1");

// firstname elements

Element firstname = doc.createElement("firstname");

firstname.appendChild(doc.createTextNode("yong"));

staff.appendChild(firstname);

// lastname elements

Element lastname = doc.createElement("lastname");

lastname.appendChild(doc.createTextNode("mook kim"));

staff.appendChild(lastname);

// nickname elements

Element nickname = doc.createElement("nickname");

nickname.appendChild(doc.createTextNode("mkyong"));

staff.appendChild(nickname);

// salary elements

Element salary = doc.createElement("salary");

salary.appendChild(doc.createTextNode("100000"));

staff.appendChild(salary);

// write the content into xml file

TransformerFactory transformerFactory = TransformerFactory.newInstance();

Transformer transformer = transformerFactory.newTransformer();

DOMSource source = new DOMSource(doc);

StreamResult result = new StreamResult(new File("C:\\file.xml"));

// Output to console for testing

// StreamResult result = new StreamResult(System.out);

transformer.transform(source, result);

System.out.println("File saved!");

} catch (ParserConfigurationException pce) {

pce.printStackTrace();

} catch (TransformerException tfe) {

tfe.printStackTrace();

}

}

}

OUTput-

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<company>

<staff id="1">

<firstname>yong</firstname>

<lastname>mook kim</lastname>

<nickname>mkyong</nickname>

<salary>100000</salary>

</staff>

</company>

**7-How to read XML file in Java – (DOM Parser)**

In this we will show you how to read an XML file via DOM XML parser. DOM parser parses the entire XML document and loads it into memory; then models it in a “TREE” structure for easy traversal or manipulation.

In short, it turns a XML file into DOM or Tree structure, and you have to traverse a node by node to get what you want.

What is Node?

In the DOM, everything in an XML document is a node, read this.

file to read:-

/Users/mkyong/staff.xml

<?xml version="1.0"?>

<company>

<staff id="1001">

<firstname>yong</firstname>

<lastname>mook kim</lastname>

<nickname>mkyong</nickname>

<salary>100000</salary>

</staff>

<staff id="2001">

<firstname>low</firstname>

<lastname>yin fong</lastname>

<nickname>fong fong</nickname>

<salary>200000</salary>

</staff>

</company>

ReadXMLFile.java-

package com.mkyong.seo;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.DocumentBuilder;

import org.w3c.dom.Document;

import org.w3c.dom.NodeList;

import org.w3c.dom.Node;

import org.w3c.dom.Element;

import java.io.File;

public class ReadXMLFile {

public static void main(String argv[]) {

try {

File fXmlFile = new File("/Users/mkyong/staff.xml");

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

Document doc = dBuilder.parse(fXmlFile);

//optional, but recommended

//read this - http://stackoverflow.com/questions/13786607/normalization-in-dom-parsing-with-java-how-does-it-work

doc.getDocumentElement().normalize();

System.out.println("Root element :" + doc.getDocumentElement().getNodeName());

NodeList nList = doc.getElementsByTagName("staff");

System.out.println("----------------------------");

for (int temp = 0; temp < nList.getLength(); temp++) {

Node nNode = nList.item(temp);

System.out.println("\nCurrent Element :" + nNode.getNodeName());

if (nNode.getNodeType() == Node.ELEMENT\_NODE) {

Element eElement = (Element) nNode;

System.out.println("Staff id : " + eElement.getAttribute("id"));

System.out.println("First Name : " + eElement.getElementsByTagName("firstname").item(0).getTextContent());

System.out.println("Last Name : " + eElement.getElementsByTagName("lastname").item(0).getTextContent());

System.out.println("Nick Name : " + eElement.getElementsByTagName("nickname").item(0).getTextContent());

System.out.println("Salary : " + eElement.getElementsByTagName("salary").item(0).getTextContent());

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

OutPut-

Root element :company

----------------------------

Current Element :staff

Staff id : 1001

First Name : yong

Last Name : mook kim

Nick Name : mkyong

Salary : 100000

Current Element :staff

Staff id : 2001

First Name : low

Last Name : yin fong

Nick Name : fong fong

Salary : 200000

**8-How to read XML file in Java – (DOM Parser)Looping the Node**

This example reads the same "staff.xml", and showing you how to loop the node one by one, and print out the node name and value, and also the attribute if any.

ReadXMLFile2.java

package com.mkyong.seo;

import java.io.File;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import org.w3c.dom.NamedNodeMap;

import org.w3c.dom.Node;

import org.w3c.dom.NodeList;

public class ReadXMLFile2 {

public static void main(String[] args) {

try {

File file = new File("/Users/mkyong/staff.xml");

DocumentBuilder dBuilder = DocumentBuilderFactory.newInstance()

.newDocumentBuilder();

Document doc = dBuilder.parse(file);

System.out.println("Root element :" + doc.getDocumentElement().getNodeName());

if (doc.hasChildNodes()) {

printNote(doc.getChildNodes());

}

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

private static void printNote(NodeList nodeList) {

for (int count = 0; count < nodeList.getLength(); count++) {

Node tempNode = nodeList.item(count);

// make sure it's element node.

if (tempNode.getNodeType() == Node.ELEMENT\_NODE) {

// get node name and value

System.out.println("\nNode Name =" + tempNode.getNodeName() + " [OPEN]");

System.out.println("Node Value =" + tempNode.getTextContent());

if (tempNode.hasAttributes()) {

// get attributes names and values

NamedNodeMap nodeMap = tempNode.getAttributes();

for (int i = 0; i < nodeMap.getLength(); i++) {

Node node = nodeMap.item(i);

System.out.println("attr name : " + node.getNodeName());

System.out.println("attr value : " + node.getNodeValue());

}

}

if (tempNode.hasChildNodes())

{

// loop again if has child nodes

printNote(tempNode.getChildNodes());

}

System.out.println("Node Name =" + tempNode.getNodeName() + " [CLOSE]");

}

}

}

}

Output:-

Root element :company

Node Name =company [OPEN]

Node Value =

yong

mook kim

mkyong

100000

low

yin fong

fong fong

200000

Node Name =staff [OPEN]

Node Value =

yong

mook kim

mkyong

100000

attr name : id

attr value : 1001

Node Name =firstname [OPEN]

Node Value =yong

Node Name =firstname [CLOSE]

Node Name =lastname [OPEN]

Node Value =mook kim

Node Name =lastname [CLOSE]

Node Name =nickname [OPEN]

Node Value =mkyong

Node Name =nickname [CLOSE]

Node Name =salary [OPEN]

Node Value =100000

Node Name =salary [CLOSE]

Node Name =staff [CLOSE]

Node Name =staff [OPEN]

Node Value =

low

yin fong

fong fong

200000

attr name : id

attr value : 2001

Node Name =firstname [OPEN]

Node Value =low

Node Name =firstname [CLOSE]

Node Name =lastname [OPEN]

Node Value =yin fong

Node Name =lastname [CLOSE]

Node Name =nickname [OPEN]

Node Value =fong fong

Node Name =nickname [CLOSE]

Node Name =salary [OPEN]

Node Value =200000

Node Name =salary [CLOSE]

Node Name =staff [CLOSE]

Node Name =company [CLOSE]**Working on Excel files**

The fundamental interfaces include Workbook, Sheet, Row and Cell. For basic formatting, use the CellStyle and Font interfaces. Concrete implementing classes include:

Excel 2003: HSSFWorkbook, HSSFSheet, HSSFRow, HSSFCell, etc.

Excel 2007: XSSFWorkbook, XSSFSheet, XSSFRow, XSSFCell, etc.

But I recommend using the common interfaces for greater flexibility with both Excel formats 2003 (XLS) and 2007(XLSX).

Here are the basic steps for writing an Excel file:-

* 1. Create a Workbook.
  2. Create a Sheet.
  3. Repeat the following steps until all data is processed:
  4. Create a Row.
  5. Create Cells in a Row. Apply formatting using CellStyle.
  6. Write to an OutputStream.
  7. Close the output stream.

Now, let’s see some examples that demonstrate writing a list of books to an Excel file.

**9-How to Write Excel Files in Java using Apache POI**

Example:-

The following code snippet is a very simple program that demonstrates writing a list of books to an Excel file in the simplest and dirty form:

package net.codejava.excel;

import java.io.FileOutputStream;

import java.io.IOException;

import org.apache.poi.ss.usermodel.Cell;

import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.xssf.usermodel.XSSFSheet;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

/\*\*

\* A very simple program that writes some data to an Excel file

\* using the Apache POI library.

\* @author www.codejava.net

\*

\*/

public class SimpleExcelWriterExample {

public static void main(String[] args) throws IOException {

XSSFWorkbook workbook = new XSSFWorkbook();

XSSFSheet sheet = workbook.createSheet("Java Books");

Object[][] bookData = {

{"Head First Java", "Kathy Serria", 79},

{"Effective Java", "Joshua Bloch", 36},

{"Clean Code", "Robert martin", 42},

{"Thinking in Java", "Bruce Eckel", 35},

};

int rowCount = 0;

for (Object[] aBook : bookData) {

Row row = sheet.createRow(++rowCount);

int columnCount = 0;

for (Object field : aBook) {

Cell cell = row.createCell(++columnCount);

if (field instanceof String) {

cell.setCellValue((String) field);

} else if (field instanceof Integer) {

cell.setCellValue((Integer) field);

}

}

}

try (FileOutputStream outputStream = new FileOutputStream("JavaBooks.xlsx")) {

workbook.write(outputStream);

}

}

}

**10-How to Write Excel Files in Java using Apache POI**

There are two main prefixes which you will encounter when working with Apache POI:

**HSSF**: denotes the API is for working with Excel 2003 and earlier.

**XSSF**: denotes the API is for working with Excel 2007 and later.

And to get started the Apache POI API, you just need to understand and use the following 4 interfaces:

Workbook: high level representation of an Excel workbook. Concrete implementations are:

HSSFWorkbook and XSSFWorkbook.

**Sheet:** high level representation of an Excel worksheet. Typical implementing classes are HSSFSheet and XSSFSheet.

**Row:** high level representation of a row in a spreadsheet. HSSFRow and XSSFRow are two concrete classes.

**Cell:** high level representation of a cell in a row. HSSFCell and XSSFCell are the typical implementing classes.

Now, let’s walk through some real-life examples.

**Reading from Excel File Examples**

Head-kathy-79

Effective-Bloach-36

clean code-robert-42

think java-bruce-35

**Code**:-

package net.codejava.excel;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.Iterator;

import org.apache.poi.ss.usermodel.Cell;

import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.ss.usermodel.Sheet;

import org.apache.poi.ss.usermodel.Workbook;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

/\*\*

\* A dirty simple program that reads an Excel file.

\* @author www.codejava.net

\*

\*/

public class SimpleExcelReaderExample {

public static void main(String[] args) throws IOException {

String excelFilePath = "Books.xlsx";

FileInputStream inputStream = new FileInputStream(new File(excelFilePath));

Workbook workbook = new XSSFWorkbook(inputStream);

Sheet firstSheet = workbook.getSheetAt(0);

Iterator<Row> iterator = firstSheet.iterator();

while (iterator.hasNext()) {

Row nextRow = iterator.next();

Iterator<Cell> cellIterator = nextRow.cellIterator();

while (cellIterator.hasNext()) {

Cell cell = cellIterator.next();

switch (cell.getCellType()) {

case Cell.CELL\_TYPE\_STRING:

System.out.print(cell.getStringCellValue());

break;

case Cell.CELL\_TYPE\_BOOLEAN:

System.out.print(cell.getBooleanCellValue());

break;

case Cell.CELL\_TYPE\_NUMERIC:

System.out.print(cell.getNumericCellValue());

break;

}

System.out.print(" - ");

}

System.out.println();

}

workbook.close();

inputStream.close();

}

}

Output:-

Head First Java - Kathy Serria - 79.0 -

Effective Java - Joshua Bloch - 36.0 -

Clean Code - Robert Martin - 42.0 -

Thinking in Java - Bruce Eckel - 35.0 -

***12-Java - OOP Concepts***

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

**Object**

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

**Class**

A class is a group of objects that has common properties. A class in java can contain:

* **Variables**
* **Method**

A class can contain any of the following variable types.

1. Local variables - Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
2. Instance variables - Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
3. Class variables - Class variables are variables declared within a class, outside any method, with the static keyword.

**Creating an Object:**

There are three steps for creating object from class.

**Declaration**: A variable declaration with a variable name with an object type.

**Instantiation**: The 'new' key word is used to create the object.

**Initialization**: The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

**Example:-**

public class Puppy {

int puppyAge;

public Puppy(String name) {

// This constructor has one parameter, name.

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

}

public void setAge( int age ) {

puppyAge = age;

}

public int getAge( ) {

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

return puppyAge;

}

public static void main(String []args) {

/\* Object creation \*/

Puppy myPuppy = new Puppy( "tommy" );

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

myPuppy.setAge( 2 );

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

myPuppy.getAge( );

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

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

}

}

**Constructors in Java**

A constructor is a special method that is used to initialize an object.Every class has a constructor,if we don't explicitly declare a constructor for any java class the compiler builds a default constructor for that class. A constructor does not have any return type.

A constructor has same name as the class in which it resides. Constructor in Java can not be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

**There are two types of Constructor**

1. Default Constructor
2. Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

**Constructor Overloading**

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.

**Example of constructor overloading**

class Cricketer

{

String name;

String team;

int age;

Cricketer () //default constructor.

{

name ="";

team ="";

age = 0;

}

Cricketer(String n, String t, int a) //constructor overloaded

{

name = n;

team = t;

age = a;

}

Cricketer (Cricketer ckt) //constructor similar to copy constructor of c++

{

name = ckt.name;

team = ckt.team;

age = ckt.age;

}

public String toString()

{

return "this is " + name + " of "+team;

}

}

Class test:

{

public static void main (String[] args)

{

Cricketer c1 = new Cricketer();

Cricketer c2 = new Cricketer("sachin", "India", 32);

Cricketer c3 = new Cricketer(c2 );

System.out.println(c2);

System.out.println(c3);

c1.name = "Virat";

c1.team= "India";

c1.age = 32;

System .out. print in (c1);

}}

**Object Array**

* Array is collection of similar data types.
* Arrays can hold collection of data with indexes
* We already know that we can hold group of primitive variables
* Arrays can hold referenced variables also like Strings and objects
* So we can say it is possible to store or create array of objects in java

We can also store custom objects in arrays:-

* Create a employee class.
* Create multiple objects of employee class and assign employee objects to array.
* Arrays can store objects but we need to instantiate each and every object and array can store it

***Example1:-***

package arraysofobjectsinjava;

public class Employee {

String name;

int id;

Employee(String name, int id)

{

this.name=name;

this.id=id;

}

public class Object\_Array

{

public static void main (String args[])

{

Employee[]emp= new Employee[5];

emp[0]=new Employee("ram",102);

emp[1]=new Employee("ram",102);

emp[2]=new Employee("ram",102);

for(Employee emp1:emp )

{

sysout(emp1.name +""+emp1.id)

}

}

}

**Inheritance**

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

An important feature of object-oriented programs is inheritance—the ability to create classes that share the attributes and methods of existing classes, but with more specific features. Inheritance is mainly used for code reusability. So you are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it’s called as Inheritance. You can look into the following example for inheritance concept. Here we have Mobile class extended by other specific class like Android and Blackberry.

**Difference between this and super Keyword**

**What is this**

This is a keyword in Java. It can be used inside the Method or constructor of Class. It(this) works as a reference to the current Object whose Method or constructor is being invoked. The this keyword can be used to refer to any member of the current object from within an instance Method or a constructor.

**This keyword with field(Instance Variable)**

this keyword can be very useful in the handling of Variable Hiding. We can not create two instance/local variables with the same name. However it is legal to create one instance variable & one local variable or Method parameter with the same name. In this scenario the local variable will hide the instance variable this is called Variable Hiding.

**Example of this keyword in java for variable hiding:-**

class JBT {

int variable = 5;

public static void main(String args[]) {

JBT obj = new JBT();

obj.method(20);

obj.method();

}

void method(int variable) {

variable = 10;

System.out.println("Value of Instance variable :" + this.variable);

System.out.println("Value of Local variable :" + variable);

}

void method() {

int variable = 40;

System.out.println("Value of Instance variable :" + this.variable);

System.out.println("Value of Local variable :" + variable);

}

}

**Output:-**

Value of Instance variable :5

Value of Local variable :10

Value of Instance variable :5

Value of Local variable :40

**Super keyword in java**

The super keyword in java 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 java super Keyword**

* super is used to refer immediate parent class instance variable.
* super() is used to invoke immediate parent class constructor.
* super is used to invoke immediate parent class method.

**Example-**

public class Person

{

int speed=10;

Person()

{

System.out.println("Person is created");

}

void message()

{

System.out.println("Person welcome");

}

}

public class PersonC extends Person

{

int speed=100;

void displaySpeed()

{

System.out.println(super.speed);//will print speed of Person now

}

PersonC()

{

super();//will invoke parent class constructor

System.out.println("PersonC is created");

}

void message()

{

System.out.println("welcome PersonC");

}

void display()

{

super.message();//will invoke parent class message() method

message();//will invoke current class message() method

}

public static void main(String args[])

{

PersonC b=new PersonC();

b.displaySpeed();

b.display();

}

}

**OutPut:-**

Person is created

PersonC is created

10

welcome PersonC

Person welcome

**Inheritance Example:-**

**Example:-**

package oopsconcept;

public class Android extends Mobile

{

//Constructor to set properties/characteristics of object

Android(String man, String o,String m, int c){

super(man, o, m, c);

}

//Method to get access Model property of Object

public String getModel(){

return "This is Android Mobile- " + model;

}

}

package oopsconcept;

public class Blackberry extends Mobile

{

//Constructor to set properties/characteristics of object

Blackberry(String man, String o,String m, int c){

super(man, o, m, c);

}

public String getModel(){

return "This is Blackberry-"+ model;

}

}

**Polymorphism**

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

**Method Overloading**

Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different. In the last tutorial we discussed constructor overloading that allows a class to have more than one constructors having different argument lists.

**Argument lists could differ in –**

1. Number of parameters.

2. Data type of parameters.

3. Sequence of Data type of parameters.

**Method overloading is also known as Static Polymorphism.**

Points to Note:

1. Static Polymorphism is also known as compile time binding or early binding.

2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

**Overloading example-**

//A class for adding upto 5 numbers

class Sum

{

int add(int n1, int n2)

{

return n1+n2;

}

int add(int n1, int n2, int n3)

{

return n1+n2+n3;

}

int add(int n1, int n2, int n3, int n4)

{

return n1+n2+n3+n4;

}

int add(int n1, int n2, int n3, int n4, int n5)

{

return n1+n2+n3+n4+n5;

}

public static void main(String args[])

{

Sum obj = new Sum();

System.out.println("Sum of two numbers: "+obj.add(20, 21));

System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));

System.out.println("Sum of four numbers: "+obj.add(20, 21, 22, 23));

System.out.println("Sum of five numbers: "+obj.add(20, 21, 22, 23, 24));

}

}

Output:

Sum of two numbers: 41

Sum of three numbers: 63

Sum of four numbers: 86

Sum of five numbers: 110

**Method Overriding**

Declaring a method in subclass which is already present in parent class is known as method overriding. Earlier we shared method overloading in java.

**Advantage of method overriding**

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method without even modifying the parent class(base class).

**Overloading vs Overriding in Java**

1. Overloading happens at [compile-time](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) while Overriding happens at [runtime](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/): The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. [Static binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overloaded methods and [dynamic binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type ([refer this](http://stackoverflow.com/questions/14694852/can-overridden-methods-differ-in-return-type)).
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

**Abstraction**

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

**Abstract Class**

**What is an Abstract Method?**

An abstract method is a method that is declared without an implementation.

It just has a method signature.

**What is an Abstract Class?**

Let’s start understanding Abstract class first and then we will go over Example.

1. An abstract class is a class that is declared abstract
2. Abstract classes cannot be instantiated
3. Abstract classes can be subclassed
4. It may or may not include abstract methods
5. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class
6. If subclass doesn’t provide implementations then the subclass must also be declared abstract.

**Can I define an abstract class without adding an abstract method?**

Of course yes. Declaring a class abstract only means that you don’t allow it to be instantiated on its own. You can’t have an abstract method in a non-abstract class.

**Example:-**

package oopsconcept;

public abstract class VehicleAbstract

{

public abstract void start();

public void stop(){

System.out.println("Stopping Vehicle in abstract class");

}

}

class TwoWheeler extends VehicleAbstract

{

@Override

public void start()

{

System.out.println("Starting Two Wheeler");

}

}

class FourWheeler extends VehicleAbstract

{

@Override

public void start()

{

System.out.println("Starting Four Wheeler");

}

}

package oopsconcept;

public class VehicleTesting {

public static void main(String[] args)

{

VehicleAbstract my2Wheeler = new TwoWheeler();

VehicleAbstract my4Wheeler = new FourWheeler();

my2Wheeler.start();

my2Wheeler.stop();

my4Wheeler.start();

my4Wheeler.stop(); } }

**Let’s start with an Example. Problem Description:**

1. Create class CrunchifyExam.java, which has one abstract method called checkResult()
2. Create class Crunchify1stSchoolExamResult.java, which extends Abstract class CrunchifyExam.java
3. Create class Crunchify2ndSchoolExamResult.java, which extends Abstract class CrunchifyExam.java
4. Now both above classes have to provide implementation for checkResult() method
5. Both Schools may have their own different procedure or number of checks to find out if user is PASSED or FAILED, they are free to have their own implementation of checkResult()

**1-CrunchifyExam.java—**

package crunchify.com.tutorial;

/\*\*

\* @author Crunchify.com

\* Simple Abstract Class and Method Example with live result

\*

\*/

public abstract class CrunchifyExam {

public enum ExamStatus {

PASSED, FAILED

}

private String examTime;

private ExamStatus status;

public CrunchifyExam(String examTime, ExamStatus status) {

this.examTime = examTime;

this.status = status;

}

public String getExamTime() {

return examTime;

}

public void setExamTime(String examTime) {

this.examTime = examTime;

}

public void setExamStatus(ExamStatus status) {

this.status = status;

}

public ExamStatus getExamStatus() {

return status;

}

abstract public void checkResult();

}

**2-Crunchify1stSchoolExamResult.java-----**

package crunchify.com.tutorial;

/\*\*

\* @author Crunchify.com

\*/

public class Crunchify1stSchoolExamResult extends CrunchifyExam {

public Crunchify1stSchoolExamResult(String examTime, ExamStatus status) {

super(examTime, status);

// TODO Auto-generated constructor stub

}

@Override

public void checkResult() {

String studentName = "Crunchify1";

String studentResult = "85%";

// School NO-1 will provide all their formula to find if user is passed or failed.

// After detailed calculation let's say student's grade is "PASSED".

System.out.println("Hey.. this is user " + studentName + " with grade " + studentResult + " - " + getExamStatus()

+ ", ExamTime: " + getExamTime());

}

}

**3-Crunchify2ndSchoolExamResult.java-----------**

package crunchify.com.tutorial;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

/\*\*

\* @author Crunchify.com

\*/

public class Crunchify2ndSchoolExamResult extends CrunchifyExam {

public Crunchify2ndSchoolExamResult(String examTime, ExamStatus status) {

super(examTime, status);

}

@Override

public void checkResult() {

String studentName = "Crunchify2";

String studentResult = "45%";

// School NO-2 will provide all their formula to find if user is passed or failed.

// After detailed calculation let's say student's grade is "FAILED".

log("Hey.. this is user " + studentName + " with grade " + studentResult + " - " + getExamStatus() + ", ExamTime: "

+ getExamTime());

}

public static void main(String args[]) {

DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");

// 1st School's checkResult()

Date date = new Date();

String examTime = dateFormat.format(date);

log("Initializing 1st School object at time " + examTime);

// We are setting up time and Result for 1st School

Crunchify1stSchoolExamResult object = new Crunchify1stSchoolExamResult(examTime, ExamStatus.PASSED);

object.checkResult();

// Let's wait 5 seconds wait to see time difference in console log

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

// 2nd School's checkResult()

date = new Date();

examTime = dateFormat.format(date);

log("\nInitializing 2nd School object at time " + examTime);

// We are setting up time and Result for 2nd School

Crunchify2ndSchoolExamResult object2 = new Crunchify2ndSchoolExamResult(examTime, ExamStatus.FAILED);

object2.checkResult();

}

// Simple log method

private static void log(String value) {

System.out.println(value);

}

}

**OutPut-Eclipse Console Result:**

Just right click on Crunchify2ndSchoolExamResult.java and run as Java Application to see below result.

Initializing 1st School object at time 2016/11/30 14:24:37

Hey.. this is user Crunchify1 with grade 85% - PASSED ExamTime: 2016/11/30 14:24:37

Initializing 2nd School object at time 2016/11/30 14:24:42

Hey.. this is user Crunchify2 with grade 45% - FAILED ExamTime: 2016/11/30 14:24:42

**Interface**

Interface is a pure abstract class.They are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body. Interface is used to achieve complete **abstraction** in Java. When you create an interface it defines what a class can do without saying anything about how the class will do it.

**Example of Interface implementation-**

interface Moveable

{

int AVG-SPEED = 40;

void move();

}

class Vehicle implements Moveable

{

public void move()

{

System .out. print in ("Average speed is"+AVG-SPEED");

}

public static void main (String[] arg)

{

Vehicle vc = new Vehicle();

vc.move();

}

}

**OutPut:**

Average speed is 40.

**Interfaces supports Multiple Inheritance-**

Though classes in java doesn't suppost multiple inheritance, but a class can implement more than one interface.

interface Moveable

{

boolean isMoveable();

}

interface Rollable

{

boolean isRollable

}

class Tyre implements Moveable, Rollable

{

int width;

boolean isMoveable()

{

return true;

}

boolean isRollable()

{

return true;

}

public static void main(String args[])

{

Tyre tr=new Tyre();

System.out.println(tr.isMoveable());

System.out.println(tr.isRollable());

}

}

**Output :**

true

true

|  |  |  |
| --- | --- | --- |
| # | **abstract Classes** | **Interfaces** |
| 1 | abstract class can extend only one class or one abstract class at a time | interface can extend any number of interfaces at a time |
| 2 | abstract  class  can extend from a class or from an abstract class | interface can extend only from an interface |
| 3 | abstract  class  can  have  both  abstract and concrete methods | interface can  have only abstract methods |
| 4 | A class can extend only one abstract class | A class can implement any number of interfaces |
| 5 | In abstract class keyword ‘abstract’ is mandatory to declare a method as an abstract | In an interface keyword ‘abstract’ is optional to declare a method as an abstract |
| 6 | abstract  class can have  protected , public and public abstract methods | Interface can have only public abstract methods i.e. by default |
| 7 | abstract class can have  static, final  or static final  variable with any access specifier | interface  can  have only static final (constant) variable i.e. by default |
| 8 | Example: public abstract class Shape{ public abstract void draw(); } | Example: public interface Drawable{ void draw(); } |

**Encapsulation**

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a “black box,” or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getXXX() methods which are having public access modifiers.

**Example:-**

package oopsconcept;

public class Mobile

{

private String manufacturer;

private String operating\_system;

public String model;

private int cost;

//Constructor to set properties/characteristics of object

Mobile(String man, String o,String m, int c){

this.manufacturer = man;

this.operating\_system=o;

this.model=m;

this.cost=c;

}

//Method to get access Model property of Object

public String getModel(){

return this.model;

}

// We can add other method to get access to other properties

}

***12.1-Java – Imported Keywords***

**Final**

The final keyword in java is used to restrict the user to do following things:-

-Stop value change

-Stop Method Overloading

-Stop Inheritance

The java final keyword can be used in many contexts. Final can be:

**Final Variable**

If you make any variable as final, you cannot change the value of final variable(It will be constant).

class Bike9{

final int speedlimit=90;//final variable

void run(){

speedlimit=400;

}

public static void main(String args[]){

Bike9 obj=new Bike9();

obj.run();

}

}//end of class

Output:Compile Time Error

**Final Method-**

If you make any method as final, you cannot override it.

class Bike{

final void run(){System.out.println("running");}

}

class Honda extends Bike{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda honda= new Honda();

honda.run();

}

}

Output:- Compile Time Error

**Class**

If you make any class as final, you cannot extend it.

final class Bike{}

class Honda1 extends Bike{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda1 honda= new Honda();

honda.run();

}

}

Output:- Compile Time Error

**Static**

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

No object needs to be created to use static variable or call static methods, just put the class name before the static variable or method to use them. Static method can not call non-static method.

**Static variable-**

Static variable is used for fulfill the common requirement. For Example company name of employees,college name of students etc. Name of the college is common for all students.

The static variable allocate memory only once in class area at the time of class loading

Suppose we want to store record of all employee of any company, in this case employee id is unique for every employee but company name is common for all. When we create a static variable as a company name then only once memory is allocated otherwise it allocate a memory space each time for every employee.

class Student

{

int roll\_no;

String name;

static String College\_Name="ITM";

}

class StaticDemo

{

public static void main(String args[])

{

Student s1=new Student();

s1.roll\_no=100;

s1.name="abcd";

System.out.println(s1.roll\_no);

System.out.println(s1.name);

System.out.println(Student.College\_Name);

Student s2=new Student();

s2.roll\_no=200;

s2.name="zyx";

System.out.println(s2.roll\_no);

System.out.println(s2.name);

System.out.println(Student.College\_Name);

}

}

**Output:**

100

abcd

ITM

200

zyx

ITM

**Static method-**

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* static method can access static data member and can change the value of it.

**There are two main restrictions for the static method. They are:**

* The static method can not use non static data member or call non-static method directly.
* this and super cannot be used in static context.

class Student9{

int rollno;

String name;

static String college = "ITS";

static void change(){

college = "BBDIT";

}

Student9(int r, String n){

rollno = r;

name = n;

}

void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){

Student9.change();

Student9 s1 = new Student9 (111,"Karan");

Student9 s2 = new Student9 (222,"Aryan");

Student9 s3 = new Student9 (333,"Sonoo");

s1.display();

s2.display();

s3.display();

}

}

**Output:**

111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

**Static block-**

Static block is a set of statements, which will be executed by the JVM before execution of main method.

At the time of class loading if we want to perform any activity we have to define that activity inside static block because this block execute at the time of class loading.

In a class we can take any number of static block but all these blocks will be execute from top to bottom

class Example3{

static int num;

static String mystr;

static{

num = 97;

mystr = "Static keyword in Java";

}

public static void main(String args[])

{

System.out.println("Value of num="+num);

System.out.println("Value of mystr="+mystr);

}

}

**Output**:

Value of num=97

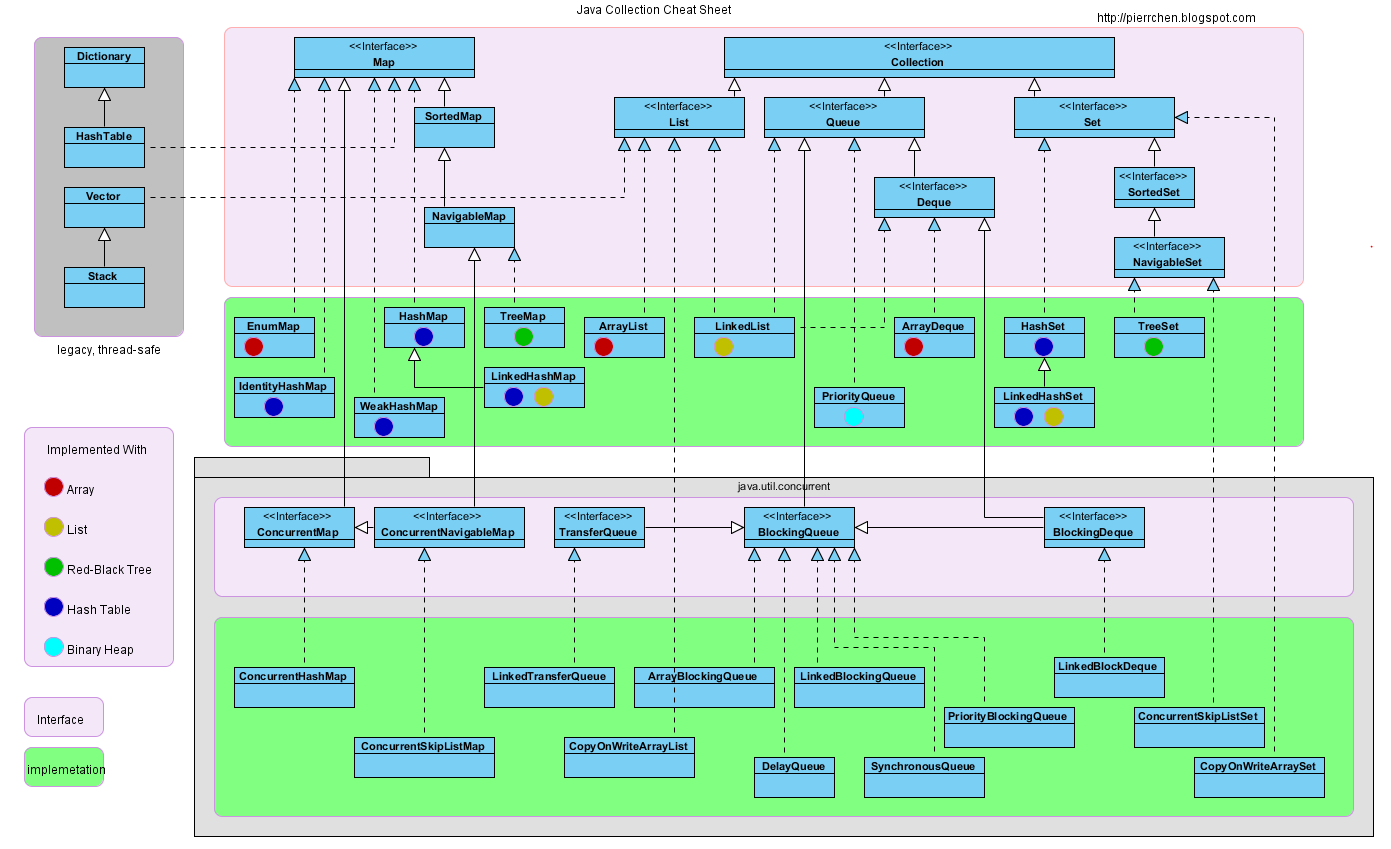
Value of mystr=Static Keyword in Java

***13-Java Collections***

**Java Collections Framework Tutorials**

The Java Collections Framework is a collection of interfaces and classes which helps in storing and processing the data efficiently. This

framework has several useful classes which have tons of useful functions which makes a programmer task super easy.



**Advantages of Collection Framework:**

* Consistent API : The API has basic set of interfaces like Collection, Set, List, or Map. All those classes (such as ArrayList, LinkedList, Vector etc) which implements, these interfaces have some common set of methods.
* Reduces programming effort: The programmer need not to worry about design of Collection rather than he can focus on its best use in his program.
* Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Iterator interface**

Iterator is used for iterating (looping) various collection classes such as [HashMap](http://beginnersbook.com/2013/12/hashmap-in-java-with-example/), [ArrayList](http://beginnersbook.com/2013/12/java-arraylist/), [LinkedList](http://beginnersbook.com/2013/12/linkedlist-in-java-with-example/) etc

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in forward direction only. |

**Methods of Iterator interface**

* There are only three methods in the Iterator interface. They are:
* public boolean hasNext() it returns true if iterator has more elements.
* public object next() it returns the element and moves the cursor pointer to the next element.
* public void remove() it removes the last elements returned by the iterator. It is rarely used.

**Iterator/ListIterator**

Both Iterator and ListIterator are used to iterate through elements of a collection class. Using Iterator we can traverse in one direction (forward) while using ListIterator we can traverse the collection class on both the directions(backward and forward).

**Iterator Example:-**

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorDemo3 {

public static void main(String args[]){

ArrayList<String> names = new ArrayList<String>();

names.add("Chaitanya");

names.add("Steve");

names.add("Jack");

Iterator<String> it = names.iterator();

while(it.hasNext()) {

String obj = it.next();

System.out.println(obj);

}

}

}

**ListIterator Example**

In this example we are traversing an ArrayList in both the directions.

import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;

public class ListIteratorExample {

public static void main(String a[]){

ListIterator<String> litr = null;

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

names.add("Shyam");

names.add("Rajat");

names.add("Paul");

names.add("Tom");

names.add("Kate");

//Obtaining list iterator

litr=names.listIterator();

System.out.println("Traversing the list in forward direction:");

while(litr.hasNext()){

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

}

System.out.println("\nTraversing the list in backward direction:");

while(litr.hasPrevious()){

System.out.println(litr.previous());

}

}

}

Output:

Traversing the list in forward direction:

Shyam

Rajat

Paul

Tom

Kate

Traversing the list in backward direction:

Kate

Tom

Paul

Rajat

Shyam

**Iterator vs ListIterator**

1)

Iterator is used for traversing List and Set both.

We can use ListIterator to traverse List only, we cannot traverse Set using ListIterator.

2)

We can traverse in only forward direction using Iterator.

Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3)

We cannot obtain indexes while using Iterator

We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

4)

We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.

We can add element at any point of time while traversing a list using ListIterator.

5)

We cannot replace the existing element value when using Iterator.

By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.

6)

**Methods of Iterator:**

hasNext()

next()

remove()

**Methods of ListIterator:**

add(E e)

hasNext()

hasPrevious()

next()

nextIndex()

previous()

previousIndex()

remove()

set(E e)

**List**

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

* [ArrayList](http://beginnersbook.com/2014/08/arraylist-in-java/)
* [LinkedList](http://beginnersbook.com/2014/08/java-linkedlist-class/)
* [Vector](http://beginnersbook.com/2014/08/java-vector-class/)

[**ArrayList**](http://beginnersbook.com/2014/08/arraylist-in-java/)**-1**

ArrayList is a resizable-array implementation of the List interface. It implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* 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.

**Example-1**

import java.util.\*;

public class LoopExample {

public static void main(String[] args) {

ArrayList<Integer> arrlist = new ArrayList<Integer>();

arrlist.add(14);

arrlist.add(7);

arrlist.add(39);

arrlist.add(40);

/\* For Loop for iterating ArrayList \*/

System.out.println("For Loop");

for (int counter = 0; counter < arrlist.size(); counter++) {

System.out.println(arrlist.get(counter));

}

/\* Advanced For Loop\*/

System.out.println("Advanced For Loop");

for (Integer num : arrlist) {

System.out.println(num);

}

/\* While Loop for iterating ArrayList\*/

System.out.println("While Loop");

int count = 0;

while (arrlist.size() > count) {

System.out.println(arrlist.get(count));

count++;

}

/\*Looping Array List using Iterator\*/

System.out.println("Iterator");

Iterator iter = arrlist.iterator();

while (iter.hasNext()) {

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

}

}

}

**Output:**

For Loop

14

7

39

40

Advanced For Loop

14

7

39

40

While Loop

14

7

39

40

Iterator

14

7

39

40

ArrayList class provides methods for basic array operations:

* add( Object o ) - puts reference to object into ArrayList
* get( int index ) - retrieves object reference from ArrayList index position
* size() - returns ArrayList size
* remove( int index ) - removes the element at the specified position in this list. Shifts any subsequent elements to the left and returns the element that was removed from the list.
* indexOf( Object o) - finds the index in this list of the first occurrence of the specified element
* clear() - removes all of the elements

**Example-2**

import java.util.\*;

public class ArrayListExamples {

public static void main(String args[]) {

// Creating an empty array list

ArrayList<String> list = new ArrayList<String>();

// Adding items to arrayList

list.add("Item1");

list.add("Item2");

list.add(2, "Item3"); // it will add Item3 to the third position of

// array list

list.add("Item4");

// Display the contents of the array list

System.out.println("The arraylist contains the following elements: "

+ list);

// Checking index of an item

int pos = list.indexOf("Item2");

System.out.println("The index of Item2 is: " + pos);

// Checking if array list is empty

boolean check = list.isEmpty();

System.out.println("Checking if the arraylist is empty: " + check);

// Getting the size of the list

int size = list.size();

System.out.println("The size of the list is: " + size);

// Checking if an element is included to the list

boolean element = list.contains("Item5");

System.out

.println("Checking if the arraylist contains the object Item5: "

+ element);

// Getting the element in a specific position

String item = list.get(0);

System.out.println("The item is the index 0 is: " + item);

// Retrieve elements from the arraylist

// 1st way: loop using index and size list

System.out

.println("Retrieving items with loop using index and size list");

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

System.out.println("Index: " + i + " - Item: " + list.get(i));

}

// 2nd way:using foreach loop

System.out.println("Retrieving items using foreach loop");

for (String str : list) {

System.out.println("Item is: " + str);

}

// 3rd way:using iterator

// hasNext(): returns true if there are more elements

// next(): returns the next element

System.out.println("Retrieving items using iterator");

for (Iterator<String> it = list.iterator(); it.hasNext();) {

System.out.println("Item is: " + it.next());

}

// Replacing an element

list.set(1, "NewItem");

System.out.println("The arraylist after the replacement is: " + list);

// Removing items

// removing the item in index 0

list.remove(0);

// removing the first occurrence of item "Item3"

list.remove("Item3");

System.out.println("The final contents of the arraylist are: " + list);

// Converting ArrayList to Array

String[] simpleArray = list.toArray(new String[list.size()]);

System.out.println("The array created after the conversion of our arraylist is: "

+ Arrays.toString(simpleArray));

}

}

In the above code, we can see that many ArrayList usage cases are covered. Adding elements to the list using 2 different methods, removing elements, getting the size of the list, checking if the list is empty, checking if a specific element is contained to the list. Also, 3 different ways are presented for retrieving the elements of a list. Finally, we show how to convert an ArrayList to Array.

**output:**

The arraylist contains the following elements: [Item1, Item2, Item3, Item4]

The index of Item2 is: 1

Checking if the arraylist is empty: false

The size of the list is: 4

Checking if the arraylist contains the object Item5: false

The item is the index 0 is: Item1

Retrieving items with loop using index and size list

Index: 0 - Item: Item1

Index: 1 - Item: Item2

Index: 2 - Item: Item3

Index: 3 - Item: Item4

Retrieving items using foreach loop

Item is: Item1

Item is: Item2

Item is: Item3

Item is: Item4

Retrieving items using iterator

Item is: Item1

Item is: Item2

Item is: Item3

Item is: Item4

The arraylist after the replacement is: [Item1, NewItem, Item3, Item4]

The final contents of the arraylist are: [NewItem, Item4]

The array created after the conversion of our arraylist is: [NewItem, Item4]

**LinkedList -2**

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* 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.

**Example-1**

import java.util.\*;

public class LinkedListExample {

public static void main(String args[]) {

/\* Linked List Declaration \*/

LinkedList<String> linkedlist = new LinkedList<String>();

/\*add(String Element) is used for adding

\* the elements to the linked list\*/

linkedlist.add("Item1");

linkedlist.add("Item5");

linkedlist.add("Item3");

linkedlist.add("Item6");

linkedlist.add("Item2");

/\*Display Linked List Content\*/

System.out.println("Linked List Content: " +linkedlist);

/\*Add First and Last Element\*/

linkedlist.addFirst("First Item");

linkedlist.addLast("Last Item");

System.out.println("LinkedList Content after addition: " +linkedlist);

/\*This is how to get and set Values\*/

Object firstvar = linkedlist.get(0);

System.out.println("First element: " +firstvar);

linkedlist.set(0, "Changed first item");

Object firstvar2 = linkedlist.get(0);

System.out.println("First element after update by set method: " +firstvar2);

/\*Remove first and last element\*/

linkedlist.removeFirst();

linkedlist.removeLast();

System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

/\* Add to a Position and remove from a position\*/

linkedlist.add(0, "Newly added item");

linkedlist.remove(2);

System.out.println("Final Content: " +linkedlist);

}

}

**OutPut:-**

Linked List Content: [Item1, Item5, Item3, Item6, Item2]

LinkedList Content after addition: [First Item, Item1, Item5, Item3, Item6, Item2, Last Item]

First element: First Item

First element after update by set method: Changed first item

LinkedList after deletion of first and last element: [Item1, Item5, Item3, Item6, Item2]

Final Content: [Newly added item, Item1, Item3, Item6, Item2]

**Doubly linked list implementation-3**

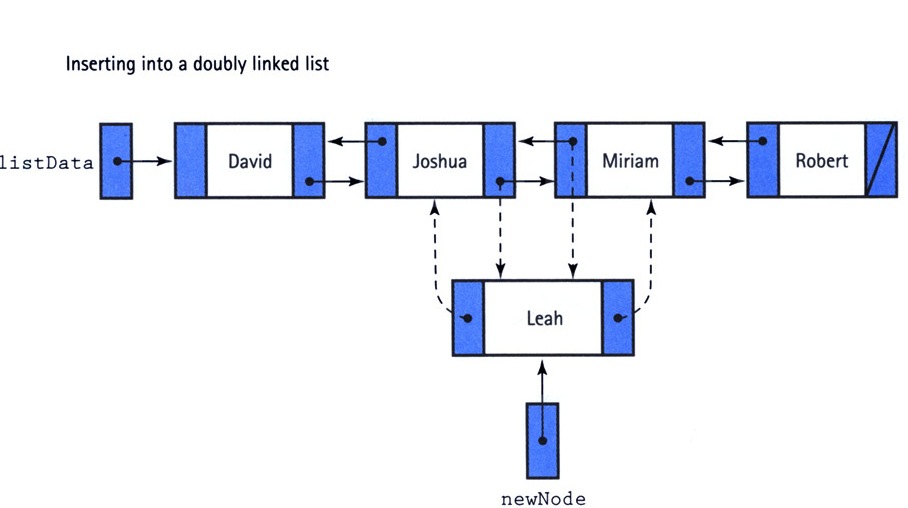
A doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

Here is the pictorial view of doubly linked list:

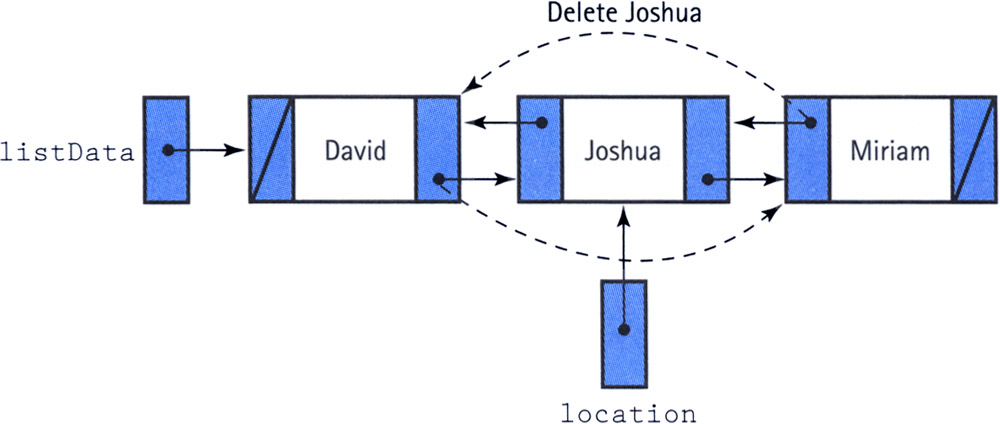
https://www.java2novice.com/images/doublell.jpg

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient, because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Here is the pictorial view of inserting an element in the middle of a doubly linked list:

Here is the pictorial view of deleting an element in the middle of a doubly linked list:



**Example:-**

package com.java2novice.ds.linkedlist;

import java.util.NoSuchElementException;

public class DoublyLinkedListImpl<E> {

private Node head;

private Node tail;

private int size;

public DoublyLinkedListImpl() {

size = 0;

}

/\*\*

\* this class keeps track of each element information

\* @author java2novice

\*

\*/

private class Node {

E element;

Node next;

Node prev;

public Node(E element, Node next, Node prev) {

this.element = element;

this.next = next;

this.prev = prev;

}

}

/\*\*

\* returns the size of the linked list

\* @return

\*/

public int size() { return size; }

/\*\*

\* return whether the list is empty or not

\* @return

\*/

public boolean isEmpty() { return size == 0; }

/\*\*

\* adds element at the starting of the linked list

\* @param element

\*/

public void addFirst(E element) {

Node tmp = new Node(element, head, null);

if(head != null ) {head.prev = tmp;}

head = tmp;

if(tail == null) { tail = tmp;}

size++;

System.out.println("adding: "+element);

}

/\*\*

\* adds element at the end of the linked list

\* @param element

\*/

public void addLast(E element) {

Node tmp = new Node(element, null, tail);

if(tail != null) {tail.next = tmp;}

tail = tmp;

if(head == null) { head = tmp;}

size++;

System.out.println("adding: "+element);

}

/\*\*

\* this method walks forward through the linked list

\*/

public void iterateForward(){

System.out.println("iterating forward..");

Node tmp = head;

while(tmp != null){

System.out.println(tmp.element);

tmp = tmp.next;

}

}

/\*\*

\* this method walks backward through the linked list

\*/

public void iterateBackward(){

System.out.println("iterating backword..");

Node tmp = tail;

while(tmp != null){

System.out.println(tmp.element);

tmp = tmp.prev;

}

}

/\*\*

\* this method removes element from the start of the linked list

\* @return

\*/

public E removeFirst() {

if (size == 0) throw new NoSuchElementException();

Node tmp = head;

head = head.next;

head.prev = null;

size--;

System.out.println("deleted: "+tmp.element);

return tmp.element;

}

/\*\*

\* this method removes element from the end of the linked list

\* @return

\*/

public E removeLast() {

if (size == 0) throw new NoSuchElementException();

Node tmp = tail;

tail = tail.prev;

tail.next = null;

size--;

System.out.println("deleted: "+tmp.element);

return tmp.element;

}

public static void main(String a[]){

DoublyLinkedListImpl<Integer> dll = new DoublyLinkedListImpl<Integer>();

dll.addFirst(10);

dll.addFirst(34);

dll.addLast(56);

dll.addLast(364);

dll.iterateForward();

dll.removeFirst();

dll.removeLast();

dll.iterateBackward();

}

}

**Output:**

adding: 10

adding: 34

adding: 56

adding: 364

iterating forward..

34

10

56

364

deleted: 34

deleted: 364

iterating backword..

56

10

**Vector-4**

The **Vector class** implements a growable array of objects. Similar to array, elements of Vector can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Vector is synchronized which means it is suitable for thread-safe operations but it gives poor performance when used in [multi-thread environment](http://beginnersbook.com/2013/03/multithreading-in-java/). It is recommended to use ArrayList (it is non-synchronized, gives good performance)  in place of Vector when there is no need of thread-safe operations

**Example-1**

import java.util.\*;

public class VectorExample {

public static void main(String args[]) {

/\* Vector of initial capacity(size) of 2 \*/

Vector<String> vec = new Vector<String>(2);

/\* Adding elements to a vector\*/

vec.addElement("Apple");

vec.addElement("Orange");

vec.addElement("Mango");

vec.addElement("Fig");

/\* check size and capacityIncrement\*/

System.out.println("Size is: "+vec.size());

System.out.println("Default capacity increment is: "+vec.capacity());

vec.addElement("fruit1");

vec.addElement("fruit2");

vec.addElement("fruit3");

/\*size and capacityIncrement after two insertions\*/

System.out.println("Size after addition: "+vec.size());

System.out.println("Capacity after increment is: "+vec.capacity());

/\*Display Vector elements\*/

Enumeration en = vec.elements();

System.out.println("\nElements are:");

while(en.hasMoreElements())

System.out.print(en.nextElement() + " ");

}

}

**Output:**

Size is: 4

Default capacity increment is: 4

Size after addition: 7

Capacity after increment is: 8

Elements are:

Apple Orange Mango Fig fruit1 fruit2 fruit3

**ArrayList vs Vector**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

**ArrayList vs LinkedList**

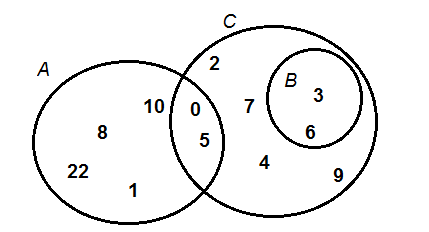
|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is 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. |
| 3) 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. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

**Array vs ArrayList**

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Arrays are static in nature. Arrays are fixed length data structures. You can’t change their size once they are created. | ArrayList is dynamic in nature. Its size is automatically increased if you add elements beyond its capacity. |
| Arrays can hold both primitives as well as objects. | ArrayList can hold only objects. |
| Arrays can be iterated only through *for* loop or *for-each* loop. | ArrayList provides iterators to iterate through their elements. |
| The size of an array is checked using *length*attribute. | The size of an ArrayList can be checked using *size()* method. |
| Array gives constant time performance for both add and get operations. | ArrayList also gives constant time performance for both add and get operations provided adding an element doesn’t trigger resize. |
| Arrays don’t support generics. | ArrayList supports generics. |
| Arrays are not type safe. | ArrayList are type safe. |
| Arrays can be multi-dimensional. | ArrayList can’t be multi-dimensional. |
| Elements are added using assignment operator. | Elements are added using add() method. |

**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. The following picture illustrates three sets of numbers 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.

**Implementation of Set**

* HashSet: is the best-performing implementation and is a widely-used Set implementation. It represents the core characteristics of sets: no duplication and unordered.
* LinkedHashSet: This implementation orders its elements based on insertion order. So consider using a LinkedHashSet when you want to store unique elements in order.
* TreeSet: This implementation orders its elements based on their values, either by their natural ordering, or by a Comparator provided at creation time.

**HashSet-1**

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:-

* HashSet stores the elements by using a mechanism called hashing.
* HashSet doesn’t maintain any order, the elements would be returned in any random order.
* HashSet doesn’t allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
* HashSet allows null values however if you insert more than one nulls it would still return only one null value.
* HashSet is non-synchronized.
* The iterator returned by this class is fail-fast which means iterator would throw ConcurrentModificationException if HashSet has been modified after creation of iterator, by any means except iterator’s own remove method.
* HashSet contains unique elements only.

**Example-1**

import java.util.\*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

this.id = id;

this.name = name;

this.author = author;

this.publisher = publisher;

this.quantity = quantity;

}

}

public class HashSetExample {

public static void main(String[] args) {

HashSet<Book> set=new HashSet<Book>();

//Creating Books

Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

//Adding Books to HashSet

set.add(b1);

set.add(b2);

set.add(b3);

//Traversing HashSet

for(Book b:set){

System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

}

}

}

**OutPut:-**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

**Example-2**

package com.javacodegeeks.core.set;

import java.util.\*;

public class SetExample {

public static void main(String args[]) {

// We create a new, empty set

Set mySet1 = new HashSet();

// We add a few elements

mySet1.add("A");

mySet1.add("C");

mySet1.add("A");

mySet1.add("B");

// Print the elements of the Set

System.out.println("mySet1: " + mySet1);

// Create a list and add some elements

List list = new ArrayList();

list.add("A");

list.add("C");

list.add("A");

list.add("A");

list.add("B");

list.add("C");

// Now create the set using the appropriate constructor

Set mySet2 = new HashSet(list);

// Print the elements of the list an the the set

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

System.out.println("mySet2: " + mySet2);

// Compare the two sets

System.out.println("MySet1 matches mySet2: " + mySet1.equals(mySet2));

// Now we will remove one element from mySet2 and compare again

mySet2.remove("A");

System.out.println("mySet2: " + mySet2);

System.out.println("MySet1 matches mySet2: " + mySet1.equals(mySet2));

// Lets check if our sets contain all the elements of the list

System.out.println("MySet1 contains all the elements: " + mySet1.containsAll(list));

System.out.println("MySet2 contains all the elements: " + mySet2.containsAll(list));

// Use of Iterator in Set

Iterator iterator = mySet1.iterator();

while (iterator.hasNext()) {

System.out.println("Iterator loop: " + iterator.next());

}

// Use of for-each in Set

for (String str : mySet1) {

System.out.println("for-each loop: " + str);

}

// Clearing all the elements

mySet1.clear();

System.out.println("mySet1 is Empty: " + mySet1.isEmpty());

// Checking the number of elements

System.out.println("mySet1 has: " + mySet1.size() + " Elements");

System.out.println("mySet2 has: " + mySet2.size() + " Elements");

// Creating an Array with the contents of the set

String[] array = mySet2.toArray(new String[mySet2.size()]);

System.out.println("The array:" + Arrays.toString(array));

}

}

**Output:**

mySet1: [A, B, C]

list: [A, C, A, A, B, C]

mySet2: [A, B, C]

MySet1 matches mySet2: true

mySet2: [B, C]

MySet1 matches mySet2: false

MySet1 contains all the elements: true

MySet2 contains all the elements: false

Iterator loop: A

Iterator loop: B

Iterator loop: C

for-each loop: A

for-each loop: B

for-each loop: C

mySet1 is Empty: true

mySet1 has: 0 Elements

mySet2 has: 2 Elements

The array:[B, C]

**LinkedHashSet-2**

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.
* HashSet doesn’t maintain any kind of order of its elements.
* TreeSet sorts the elements in ascending order.
* LinkedHashSet maintains the insertion order. Elements gets sorted in the same sequence in which they have been added to the Set.

**Example**

import java.util.\*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

this.id = id;

this.name = name;

this.author = author;

this.publisher = publisher;

this.quantity = quantity;

}

}

public class LinkedHashSetExample {

public static void main(String[] args) {

LinkedHashSet<Book> hs=new LinkedHashSet<Book>();

//Creating Books

Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

//Adding Books to hash table

hs.add(b1);

hs.add(b2);

hs.add(b3);

//Traversing hash table

for(Book b:hs){

System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

}

}

}

**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

**TreeSet-3**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order. TreeSet allows null element but like HashSet it doesn’t allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order

**Example:-**

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement Comparable interface.

import java.util.\*;

class Book implements Comparable<Book>{

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

this.id = id;

this.name = name;

this.author = author;

this.publisher = publisher;

this.quantity = quantity;

}

public int compareTo(Book b) {

if(id>b.id){

return 1;

}else if(id<b.id){

return -1;

}else{

return 0;

}

}

}

public class TreeSetExample {

public static void main(String[] args) {

Set<Book> set=new TreeSet<Book>();

//Creating Books

Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);

Book b2=new Book(233,"Operating System","Galvin","Wiley",6);

Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

//Adding Books to TreeSet

set.add(b1);

set.add(b2);

set.add(b3);

//Traversing TreeSet

for(Book b:set){

System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

}

}

}

**Output:**

101 Data Communications & Networking Forouzan Mc Graw Hill 4

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6

**Hashset vs HashMap**

|  |  |
| --- | --- |
| **HashSet** | **HashMap** |
| HashSet implements Set interface. | HashMap implements Map interface. |
| HashSet stores the data as objects. | HashMap stores the data as key-value pairs. |
| HashSet internally uses HashMap. | HashMap internally uses an array of Entry<K, V> objects. |
| HashSet doesn’t allow duplicate elements. | HashMap doesn’t allow duplicate keys, but allows duplicate values. |
| HashSet allows only one null element. | HashMap allows one null key and multiple null values. |
| Insertion operation requires only one object. | Insertion operation requires two objects, key and value. |
| HashSet is slightly slower than HashMap. | HashMap is slightly faster than HashSet. |

**HashMap vs Hashtable**

|  |  |
| --- | --- |
| **HashSet** | **HashTable** |
| HashMap is not synchronized and therefore it is not thread safe. | HashTable is internally synchronized and therefore it is thread safe. |
| HashMap allows maximum one null key and any number of null values. | HashTable doesn’t allow null keys and null values. |
| Iterators returned by the HashMap are fail-fast in nature. | Enumeration returned by the HashTable are fail-safe in nature. |
| HashMap extends AbstractMap class. | HashTable extends Dictionary class. |
| HashMap returns only iterators to traverse. | HashTable returns both Iterator as well as Enumeration for traversal. |
| HashMap is fast | HashTable is slow. |
| HashMap is not a legacy class. | HashTable is a legacy class. |
| HashMap is preferred in single threaded applications. If you want to use HashMap in multi threaded application, wrap it using Collections.synchronizedMap() method. | Although HashTable is there to use in multi threaded applications, now a days it is not at all preferred. Because, ConcurrentHashMap is better option than HashTable. |

**HashSet Vs LinkedHashSet Vs TreeSet**

|  |  |  |  |
| --- | --- | --- | --- |
| **Points** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

**Map**

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

* Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
* Several methods throw a NoSuchElementException when no items exist in the invoking map.
* A ClassCastException is thrown when an object is incompatible with the elements in a map.
* A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
* An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

A Map is an object that maps keys to values. A map cannot contain duplicate keys. There are three main implementations of Map interfaces:

**HashMap, TreeMap, and LinkedHashMap:-**

* HashMap: it makes no guarantees concerning the order of iteration
* TreeMap: It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap.
* LinkedHashMap: It orders its elements based on the order in which they were inserted into the set (insertion-order).

**Useful methods of Map interface:-**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object put(Object key, Object value) | It is used to insert an entry in this map. |
| void putAll(Map map) | It is used to insert the specified map in this map. |
| Object remove(Object key) | It is used to delete an entry for the specified key. |
| Object get(Object key) | It is used to return the value for the specified key. |
| pboolean containsKey(Object key) | It is used to search the specified key from this map. |
| Set keySet() | It is used to return the Set view containing all the keys. |
| Set entrySet() | It is used to return the Set view containing all the keys and values. |

**Map.Entry Interface**

Entry is the sub interface of Map. So we will be accessed it by Map. Entry name. It provides methods to get key and value.

Methods of Map.Entry interface.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object getKey() | It is used to obtain key. |
| Object getValue() | It is used to obtain value. |

**HashMap-1**

Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

**The important points about Java HashMap are:**

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.
* It is a Map based collection class that is used for storing Key & value pairs. This class makes no guarantees as to the order of the map.
* It is similar to the Hashtable class except that it is unsynchronized.

**HashMap Class Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| Object put(Object key, Object value) | It is used to associate the specified value with the specified key in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

**Example:-1**

package beginnersbook.com;

import java.util.HashMap;

import java.util.Map;

import java.util.Iterator;

import java.util.Set;

public class Details {

public static void main(String args[]) {

/\* This is how to declare HashMap \*/

HashMap<Integer, String> hmap = new HashMap<Integer, String>();

/\*Adding elements to HashMap\*/

hmap.put(12, "Chaitanya");

hmap.put(2, "Rahul");

hmap.put(7, "Singh");

hmap.put(49, "Ajeet");

hmap.put(3, "Anuj");

/\* Display content using Iterator\*/

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry mentry = (Map.Entry)iterator.next();

System.out.print("key is: "+ mentry.getKey() + " & Value is: ");

System.out.println(mentry.getValue());

}

/\* Get values based on key\*/

String var= hmap.get(2);

System.out.println("Value at index 2 is: "+var);

/\* Remove values based on key\*/

hmap.remove(3);

System.out.println("Map key and values after removal:");

Set set2 = hmap.entrySet();

Iterator iterator2 = set2.iterator();

while(iterator2.hasNext()) {

Map.Entry mentry2 = (Map.Entry)iterator2.next();

System.out.print("Key is: "+mentry2.getKey() + " & Value is: ");

System.out.println(mentry2.getValue());

}

}

}

**Output:**

key is: 49 & Value is: Ajeet

key is: 2 & Value is: Rahul

key is: 3 & Value is: Anuj

key is: 7 & Value is: Singh

key is: 12 & Value is: Chaitanya

Value at index 2 is: Rahul

Map key and values after removal:

Key is: 49 & Value is: Ajeet

Key is: 2 & Value is: Rahul

Key is: 7 & Value is: Singh

Key is: 12 & Value is: Chaitanya

**Example-2 Books**

import java.util.\*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

this.id = id;

this.name = name;

this.author = author;

this.publisher = publisher;

this.quantity = quantity;

}

}

public class MapExample {

public static void main(String[] args) {

//Creating map of Books

Map<Integer,Book> map=new HashMap<Integer,Book>();

//Creating Books

Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

//Adding Books to map

map.put(1,b1);

map.put(2,b2);

map.put(3,b3);

//Traversing map

for(Map.Entry<Integer, Book> entry:map.entrySet()){

int key=entry.getKey();

Book b=entry.getValue();

System.out.println(key+" Details:");

System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

}

}

}

**Output:**

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

**TreeMap -2**

TreeMap is Red-Black tree based NavigableMap implementation. It is sorted according to the natural ordering of its keys.

The important points about Java TreeMap class are:

* It implements Map interface similar to HashMap class.
* The main difference between them is that HashMap is an unordered collection while TreeMap is sorted in the ascending order of its keys.
* It is unsynchronized collection class which means it is not suitable for thread-safe operations until unless synchronized explicitly.
* It implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.
* It contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

**Methods of Java TreeMap class-**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| Object firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| Object lastKey() | It is used to return the last (highest) key currently in this sorted map. |
| Object remove(Object key) | It is used to remove the mapping for this key from this TreeMap if present. |
| void putAll(Map map) | It is used to copy all of the mappings from the specified map to this map. |
| Set entrySet() | It is used to return a set view of the mappings contained in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

**Example-1**

import java.util.TreeMap;

import java.util.Set;

import java.util.Iterator;

import java.util.Map;

public class Details {

public static void main(String args[]) {

/\* This is how to declare TreeMap \*/

TreeMap<Integer, String> tmap =

new TreeMap<Integer, String>();

/\*Adding elements to TreeMap\*/

tmap.put(1, "Data1");

tmap.put(23, "Data2");

tmap.put(70, "Data3");

tmap.put(4, "Data4");

tmap.put(2, "Data5");

/\* Display content using Iterator\*/

Set set = tmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry mentry = (Map.Entry)iterator.next();

System.out.print("key is: "+ mentry.getKey() + " & Value is: ");

System.out.println(mentry.getValue());

}

}

}

**Output:**

key is: 1 & Value is: Data1

key is: 2 & Value is: Data5

key is: 4 & Value is: Data4

key is: 23 & Value is: Data2

key is: 70 & Value is: Data3

**Example-2**

import java.util.\*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

this.id = id;

this.name = name;

this.author = author;

this.publisher = publisher;

this.quantity = quantity;

}

}

public class MapExample

{

public static void main(String[] args) {

//Creating map of Books

Map<Integer,Book> map=new TreeMap<Integer,Book>();

//Creating Books

Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

//Adding Books to map

map.put(2,b2);

map.put(1,b1);

map.put(3,b3);

//Traversing map

for(Map.Entry<Integer, Book> entry:map.entrySet()){

int key=entry.getKey();

Book b=entry.getValue();

System.out.println(key+" Details:");

System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

}

}

}

**Output:**

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

**LinkedHashMap-3**

It is a Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

**The important points about Java LinkeHashMap class are:**

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.
* HashMap doesn’t maintain any order.
* TreeMap sort the entries in ascending order of keys.
* LinkedHashMap maintains the insertion order.

**Example-1**

import java.util.LinkedHashMap;

import java.util.Set;

import java.util.Iterator;

import java.util.Map;

public class LinkedHashMapDemo {

public static void main(String args[]) {

// HashMap Declaration

LinkedHashMap<Integer, String> lhmap =

new LinkedHashMap<Integer, String>();

//Adding elements to LinkedHashMap

lhmap.put(22, "Abey");

lhmap.put(33, "Dawn");

lhmap.put(1, "Sherry");

lhmap.put(2, "Karon");

lhmap.put(100, "Jim");

// Generating a Set of entries

Set set = lhmap.entrySet();

// Displaying elements of LinkedHashMap

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry me = (Map.Entry)iterator.next();

System.out.print("Key is: "+ me.getKey() +

"& Value is: "+me.getValue()+"\n");

}

}

}

**Output:**

Key is: 22& Value is: Abey

Key is: 33& Value is: Dawn

Key is: 1& Value is: Sherry

Key is: 2& Value is: Karon

Key is: 100& Value is: Jim

**Example-2**

import java.util.\*;

class Book

{

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

this.id = id;

this.name = name;

this.author = author;

this.publisher = publisher;

this.quantity = quantity;

}

}

public class MapExample

{

public static void main(String[] args) {

//Creating map of Books

Map<Integer,Book> map=new LinkedHashMap<Integer,Book>();

//Creating Books

Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

//Adding Books to map

map.put(2,b2);

map.put(1,b1);

map.put(3,b3);

//Traversing map

for(Map.Entry<Integer, Book> entry:map.entrySet()){

int key=entry.getKey();

Book b=entry.getValue();

System.out.println(key+" Details:");

System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

}

}

}

**Output:**

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

3 Details:

103 Operating System Galvin Wiley 6

**LinkedHashMap vs TreeMap vs HashMap**

|  |  |  |  |
| --- | --- | --- | --- |
| **s** | **HashMap** | **LinkedHashMap** | **TreeMap** |
|  | Extends AbstractMap class. | Extends HashMap class. | Extends AbstractMap class. |
|  | Implements Map interface. | Implements Map interface. Implementation of Hash Table and Linked List. | Implements NavigableMap and SortedMap interfaces. |
|  | Provide constant time performance for ‘get’ and ‘put’ operations. | Maintains a doubly-linked list | Underlying data structure is Red-Black Tree. |
|  | No Iteration Order. | Iteration takes place in the order in which the entries were put into the map. | Insertion order is not maintained. Sorted according to the natural order of keys. |
|  | Key can be null. | Value can be null. | Key can’t be null; if it’s null, java.lang.NullPointerException occurs. |
|  | Value can be null. | Value can be null. | Value can be null. |
|  | Fail-fast. | Fail-fast. | Fail-fast behavior is not guaranteed always. |
|  | Duplicate keys are not allowed. | Duplicate keys are not allowed. | Duplicate keys are not allowed. |
|  | Implementation is not synchronized. | Implementation is not synchronized. | Implementation is not synchronized. |
| Ordering and Sorting | HashMap doesn't provide any ordering guarantee for entries, which means, you can not assume any order while iterating over keys and values of HashMap. This behavior of HashMap is similar to Hashtable while other two Map implementation provides ordering guarantee. | LinkedHashMap can be used to maintain insertion order, on which keys are inserted into Map or it can also be used to maintain an access order, on which keys are accessed. This provides LinkedHashMap an edge over HashMap without compromising too much performance | TreeMap provides you complete control over sorting elements by passing custom Comparator of your choice, but with the expense of some performance. Since entries are stored in a tree-based data structure, it provides lower performance than HashMap and LinkedHashMap. |
| Null keys and Values | HashMap allows one null key and multiple null values. It keeps null key based entries on index[0] on an internal bucket. If you look at the put() method of HashMap, you can see, it doesn't throw NullPointerException for null keys | LinkedHashMap is a subclass of HashMap, it also allows null keys and values. | TreeMap, which sorts elements in natural order doesn't allow null keys because compareTo() method throws NullPointerException if compared with null. If you are using TreeMap with user defined Comparator than it depends upon the implementation of compare() method. |

**Differences between List, Set, Map**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Property** | **java.util.List** | **java.util.Set** | **java.util.Map** |
| 1 | Duplicate elements | List **allows to store duplicate elements** in java. | Set does ***not allow to store duplicate elements*** in java*.* | Map stores data in form of ***key-value pair*** *it does not allow to store duplicate keys but allows duplicate values* in java*.* |
| 2 | Insertion order | java.util.List is ordered collection it **maintain insertion order** in java. | Most of the java.util.Set implementation does not maintain insertion order. HashSet does not maintains insertion order in java. Thought LinkedHashSet maintains insertion order in java. TreeSet is sorted by natural order in java. | Most of the java.util.Map implementation does not maintain insertion order. HashMap does not maintains insertion order in java. Thought LinkedHashMap maintains insertion order of keys in java. TreeMap is sorted by natural order of keys in java. |
| 3 | Null keys | List allows to store **many null keys** in java. | Most of the Set implementations allow to add only one null in java. TreeSet and ConcurrentSkipListSet does not allow to add null in java. | Lets look at Map implementations - HashMap allows one null key and many null values. LinkedHashMap allows one null key and many null values. TreeMap doesn't allow null key but allow many null values. Hashtable doesn't allow null key or null values. ConcurrentHashMap doesn't allow null key or null values. ConcurrentSkipListMap doesn't allow null key or null values. |
| 4 | Getting element on specific **index** | List implementations provide get method to get element on specific index in java.  ArrayList, Vector, copyOnWriteArrayList and LinkedList provides - get(int index) Method returns element on specified index. Get method directly gets element on specified index. | Set implementations does not provide any such get method to get element on specified index in java. | Map implementations does not provide any such get method to get element on specified index in java. |
| 5 | Implementing classes | ArrayList, LinkedList, Vector, CopyOnWriteArrayList classes implements List interface in java. | HashSet, CopyOnWriteArraySet, LinkedHashSet, TreeSet, ConcurrentSkipListSet, EnumSet classes implements Set interface in java. | HashMap, Hashtable, ConcurrentHashMap,  LinkedHashMap,  TreeMap,  ConcurrentSkipListMap,  IdentityHashMap,WeakHashMap,  EnumMap classes implements Map interface in java. |
| 6 | listIterator | listIterator method returns listIterator to iterate over elements in List in java. listIterator provides additional methods as compared to iterator like hasPrevious(), previous(), nextIndex(), previousIndex(), add(E element), set(E element) | Set does not provide anything like listIterator. It simply return Iterator in java. | Map provides three type of iterators: - map.keySet().iterator() method returns iterator to iterate over keys in HashMap map.values().iterator() method returns iterator to iterate over keys in HashMap in java. map.entrySet().iterator() method returns iterator to iterate over keys in HashMap. |
| 7 | Structure and resizable | List are Resizable-array implementation of the java.util.**List** interface in java. | Set uses Map for their implementation. Hence, structure is map based and resizing depends on Map implementation. Example > HashSet internally uses HashMap. | [Map uses hashing technique for storing key-value pairs.](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) |
| 8 | Index based structure /RandomAccess | As ArrayList uses array for implementation it is index based structure, hence provides random access to elements.But LinkedList is not indexed based structure in java. | Set is not index based structure at all in java. | Map is not index based structure at all in java. |
| 9 | unsynchronized implementations | ArrayList, LinkedList | HashSet, LinkedHashSet, TreeSet, EnumSet | HashMap,  LinkedHashMap,  TreeMap,  IdentityHashMap, WeakHashMap,  EnumMap |
| 10 | synchronized implementations | Vector, CopyOnWriteArrayList | CopyOnWriteArraySet, ConcurrentSkipListSet | Hashtable, ConcurrentHashMap, ConcurrentSkipListMap, |