# ABOUT THE TUTORIAL

## Java Tutorial

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

This reference will take you through simple and practical approach while learning Java Programming language.



## Audience

This reference has been prepared for the beginners to help them understand the basic to advanced concepts related to Java Programming language.

## Prerequisites

Before you start doing practice with various types of examples given in this reference, I'm making an assumption that you are already aware about what is a computer program and what is a computer programming language?

## Copyright & Disclaimer Notice

CHAPTER

2

# Java Environment Setup

Before we proceed further, it is important that we set up the Java environment correctly. This section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

Java SE is freely available from the link Download Java. So you download a version based on your operating system.

Follow the instructions to download Java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:



## Setting up the path for Windows 2000/XP:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

## Setting up the path for windows 95/98/ME:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

- Edit the 'C:\autoexec.bat' file and add the following line at the end: 'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

## Setting up the path for Linux, UNIX, Solaris, FreeBSD:

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:$PATH'

# Classes in Java:

A class is a blue print from which individual objects are created.

A sample of a class is given below:

```
public class Dog{
String breed;
int age;
String color;

void barking(){
}

void hungry(){
}

void sleeping(){
}
}
```

A class can contain any of the following variable types.

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



A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

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

# Constructors:

When discussing about classes, one of the most important subtopic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Example of a constructor is given below:

```
public class Puppy{
public Puppy(){
}

public Puppy(String name){
// This constructor has one parameter, name.
}
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

```
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

# Example:

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

```
public class Puppy{

int puppyAge;

public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name 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 =newPuppy("tommy");

/* Call class method to set puppy's age */
myPuppy.setAge(2);

/* Call another class method to get puppy's age */
  myPuppy.getAge();

/* You can access instance variable as follows as well */
System.out.println("Variable Value :"+ myPuppy.puppyAge );
}
}
```



If we compile and run the above program, then it would produce the following result:

```
PassedName is:tommy
Puppy's age is :2
Variable Value :2
```

# Source file declaration rules:

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non public classes.
- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example : The class name is . *public class Employee{}* Then the source file should be as Employee.java.

```
Salary:500.0
```

## What is Next?

Next session will discuss basic data types in Java and how they can be used when developing Java applications.

# Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a =68;
char a ='A'
```

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal=100;
int octal =0144;
int hexa =0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a ='\u0001';
String a ="\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

| Notation | Character represented |
|---|---|
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | Tab |
| \" | Double quote |
| \' | Single quote |
| \\ | Backslash |
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

CHAPTER

6

# Java Variable Types

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

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

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

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

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```

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

- Local variables
- Instance variables
- Class/static variables

## Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.

# Private Access Modifier - private:

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

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

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

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

## Example:

The following class uses private access control:

```
public class Logger{
private String format;
public String getFormat(){
return this.format;
}
  public void setFormat(String format){
this.format = format;
}
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.



# Public Access Modifier - public:

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

However if the public class we are trying to access is in a different package, then the public class still need to be imported.

Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

## Example:

The following function uses public access control:

```
public static void main(String[] arguments){
// ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

# Protected Access Modifier - protected:

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

```
}
```

This produces the following result:

```
false
true
false
```

# valueOf()

## Description:

The valueOf method returns the relevant Number Object holding the value of the argument passed. The argument can be a primitive data type, String, etc.

This method is a static method. The method can take two arguments, where one is a String and the other is a radix.

## Syntax:

All the variants of this method are given below:

```
staticInteger valueOf(int i)
staticInteger valueOf(String s)
staticInteger valueOf(String s, int radix)
```



## Parameters:

Here is the detail of parameters:

- **i** -- An int for which Integer representation would be returned.
- **s** -- A String for which Integer representation would be returned.
- **radix** -- This would be used to decide the value of returned Integer based on passed String.

## Return Value:

- **valueOf(int i):** This returns an Integer object holding the value of the specified primitive.
- **valueOf(String s):** This returns an Integer object holding the value of the specified string representation.
- **valueOf(String s, int radix):** This returns an Integer object holding the integer value of the specified string representation, parsed with the value of radix.

```
public class Test{

public static void main(String args[]){

Integer x =Integer.valueOf(9);
Double c =Double.valueOf(5);
Float a =Float.valueOf("80");

Integer b =Integer.valueOf("444",16);

System.out.println(x);
```

- A primitive data types

## Return Value:

- This method returns the maximum of the two arguments.

## Example:

```
public class Test{

public static void main(String args[]){
System.out.println(Math.max(12.123,12.456));
System.out.println(Math.max(23.12,23.0));
}
}
```

This produces the following result:

```
12.456
23.12
```

# exp()

## Description:

The method returns the base of the natural logarithms, e, to the power of the argument.



## Syntax:

```
double exp(double d)
```

## Parameters:

Here is the detail of parameters:

- **d** -- A primitive data types

## Return Value:

- This method Returns the base of the natural logarithms, e, to the power of the argument.

## Example:

```
public class Test{

public static void main(String args[]){
double x =11.635;
double y =2.76;

System.out.printf("The value of e is %.4f%n",Math.E);
```

```
System.out.printf("exp(%.3f) is %.3f%n", x,Math.exp(x));
}
}
```

This produces the following result:

```
The value of e is 2.7183
exp(11.635) is 112983.831
```

# log()

## Description:

The method returns the natural logarithm of the argument.

## Syntax:

```
double log(double d)
```

## Parameters:

Here is the detail of parameters:

- **d** -- A primitive data types

## Return Value:

- This method Returns the natural logarithm of the argument.



## Example:

```
public class Test{

public static void main(String args[]){
double x =11.635;
double y =2.76;

System.out.printf("The value of e is %.4f%n",Math.E);
System.out.printf("log(%.3f) is %.3f%n", x,Math.log(x));
}
}
```

This produces the following result:

```
The value of e is 2.7183
log(11.635) is 2.454
```

# pow()

## Description:

```
double radians =Math.toRadians(degrees);

System.out.format("The value of pi is %.4f%n",Math.PI);
System.out.format("The sine of %.1f degrees is
%.4f%n",degrees,Math.sin(radians));

}
}
```

This produces the following result:

```
The value of pi is 3.1416
The sine of 45.0 degrees is 0.7071
```

# cos()

## Description:

The method returns the cosine of the specified double value.

## Syntax:

```
double cos(double d)
```



## Parameters:

Here is the detail of parameters:

- **d** -- A double data types

## Return Value:

- This method Returns the cosine of the specified double value.

## Example:

```
public class Test{

public static void main(String args[]){
double degrees =45.0;
double radians =Math.toRadians(degrees);

System.out.format("The value of pi is %.4f%n",Math.PI);
System.out.format("The cosine of %.1f degrees is %.4f%n",
                        degrees,Math.cos(radians));

}
}
```

This produces the following result:

```
The value of pi is 3.1416
```

- **d** -- A double data types

## Return Value:

- This method Returns the arccosine of the specified double value.

## Example:

```
public class Test{

public static void main(String args[]){
double degrees =45.0;
double radians =Math.toRadians(degrees);

System.out.format("The value of pi is %.4f%n",Math.PI);
System.out.format("The arccosine of %.4f is %.4f degrees %n",
Math.cos(radians),
Math.toDegrees(Math.acos(Math.sin(radians))));


}
}
```

This produces the following result:

```
The value of pi is 3.1416
The arccosine of 0.7071 is 45.0000 degrees
```



# atan()

## Description:

The method returns the arctangent of the specified double value.

## Syntax:

```
double atan(double d)
```

## Parameters:

Here is the detail of parameters:

- **d** -- A double data types

## Return Value :

- This method Returns the arctangent of the specified double value.

## Example:

```
}
```

This produces the following result:

```
0.982793723247329
```

# toDegrees()

## Description:

The method converts the argument value to degrees.

## Syntax:

```
double toDegrees(double d)
```

## Parameters:

Here is the detail of parameters:

- **d** -- A double data type.

## Return Value:

- This method returns a double value.



## Example:

```
public class Test{

public static void main(String args[]){
double x =45.0;
double y =30.0;

System.out.println(Math.toDegrees(x));
System.out.println(Math.toDegrees(y));
}
}
```

This produces the following result:

```
2578.3100780887044
1718.8733853924698
```

# toRadians()

## Description:

The method converts the argument value to radians.

CHAPTER

12

# Java Characters

Normally, when we work with characters, we use primitive data types char.

## Example:

```
char ch ='a';

// Unicode for uppercase Greek omega character
char uniChar ='\u039A';

// an array of chars
char[] charArray ={'a','b','c','d','e'};
```



However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides wrapper class **Character** for primitive data type char.

The Character class offers a number of useful class (i.e., static) methods for manipulating characters. You can create a Character object with the Character constructor:

```
Character ch =newCharacter('a');
```

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This feature is called autoboxing or unboxing, if the conversion goes the other way.

## Example:

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch ='a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

## Escape Sequences:

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.

# isDigit()

## Description:

The method determines whether the specified char value is a digit.

## Syntax:

```
boolean isDigit(char ch)
```

## Parameters:

Here is the detail of parameters:

- **ch** -- Primitive character type

## Return Value:

- This method Returns true if passed character is really a digit.

## Example:

```
public class Test{

public static void main(String args[]){
System.out.println(Character.isDigit('c'));
System.out.println(Character.isDigit('5'));
}
}
```

This produces the following result:

```
false
true
```

# isWhitespace()

## Description:

The method determines whether the specified char value is a white space, which includes space, tab or new line.

## Syntax:

```
boolean isWhitespace(char ch)
```

This produces the following result:

```
Returned String: hello world
Returned String: llo wo
```

# boolean endsWith(String suffix)

## Description:

This method tests if this string ends with the specified suffix.

## Syntax:

Here is the syntax of this method:

```
public boolean endsWith(String suffix)
```

## Parameters:

Here is the detail of parameters:

- **suffix** -- the suffix.

## Return Value:

- This method returns true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise. Note that the result will be true if the argument is the empty string or is equal to this String object as determined by the equals(Object) method.



## Example:

```
public class Test{

public static void main(String args[]){
String Str=new String("This is really not immutable!!");
boolean retVal;

    retVal =Str.endsWith("immutable!!");
System.out.println("Returned Value = "+ retVal );

    retVal =Str.endsWith("immu");
System.out.println("Returned Value = "+ retVal );
}
}
```

This produces the following result:

```
Returned Value = true
Returned Value = false
```

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

## Syntax:

Here is the syntax of this method:

```
public boolean equalsIgnoreCase(String anotherString)
```

## Parameters:

Here is the detail of parameters:

- **anotherString** -- the String to compare this String against

## Return Value:

- This method returns true if the argument is not null and the Strings are equal, ignoring case; false otherwise.

## Example:



```
public class Test{

public static void main(String args[])
String Str1=new String("This is really not immutable!!");
String Str2=Str1;
String Str3=new String("This is really not immutable!!");
String Str4=new String("This IS REALLY NOT IMMUTABLE!!");
boolean retVal;

    retVal =Str1.equals(Str2);
System.out.println("Returned Value = "+ retVal );

    retVal =Str1.equals(Str3);
System.out.println("Returned Value = "+ retVal );

    retVal =Str1.equalsIgnoreCase(Str4);
System.out.println("Returned Value = "+ retVal );
}
}
```

This produces the following result:

```
Returned Value = true
Returned Value = true
Returned Value = true
```

# byte getBytes()

## Description:

```
String Str=new String("Welcome to Tutorialspoint.com");
StringSubStr1=new String("Tutorials");
StringSubStr2=new String("Sutorials");

System.out.print("Found Index :");
System.out.println(Str.indexOf('o'));
System.out.print("Found Index :");
System.out.println(Str.indexOf('o',5));
System.out.print("Found Index :");
System.out.println(Str.indexOf(SubStr1));
System.out.print("Found Index :");
System.out.println(Str.indexOf(SubStr1,15));
System.out.print("Found Index :");
System.out.println(Str.indexOf(SubStr2));
}
}
```

This produces the following result:

```
Found Index :4
Found Index :9
Found Index :11
Found Index :-1
Found Index :-1
```



# int indexOf(String str, int fromIndex)

## Description:

This method has following different variants:

- **public int indexOf(int ch):** Returns the index within this string of the first occurrence of the specified character or -1 if the character does not occur.
- **public int indexOf(int ch, int fromIndex):** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index or -1 if the character does not occur.
- **int indexOf(String str):** Returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.
- **int indexOf(String str, int fromIndex):** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. If it does not occur, -1 is returned.

## Syntax:

Here is the syntax of this method:

```
public int indexOf(int ch )

or

public int indexOf(int ch,int fromIndex)

or

int indexOf(String str)

or
```

- **public int lastIndexOf(String str, int fromIndex):** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

## Syntax:

Here is the syntax of this method:

```
int lastIndexOf(int ch)

or

public int lastIndexOf(int ch,int fromIndex)

or

public int lastIndexOf(String str)

or

public int lastIndexOf(String str,int fromIndex)
```

## Parameters:

Here is the detail of parameters:

- **ch** -- a character.
- **fromIndex** -- the index to start the search from.
- **str** -- A string.



## Return Value:

- This method returns the index.

## Example:

```
import java.io.*;

public class Test{

public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");
String SubStr1=new String("Tutorials");
String SubStr2=new String("Sutorials");

System.out.print("Found Last Index :");
System.out.println(Str.lastIndexOf('o'));
System.out.print("Found Last Index :");
System.out.println(Str.lastIndexOf('o',5));
System.out.print("Found Last Index :");
System.out.println(Str.lastIndexOf(SubStr1));
System.out.print("Found Last Index :");
System.out.println(Str.lastIndexOf(SubStr1,15));
System.out.print("Found Last Index :");
System.out.println(Str.lastIndexOf(SubStr2));
}
```

```
}
System.out.println("");
System.out.println("Return Value :");
for(String retval:Str.split("-")){
System.out.println(retval);
}
}
}
```

This produces the following result:

```
Return Value :
Welcome
to-Tutorialspoint.com

Return Value :
Welcome
to
Tutorialspoint.com

Return Value:
Welcome
to
Tutorialspoint.com

Return Value :
Welcome
to
Tutorialspoint.com
```



# boolean startsWith(String prefix)

## Description:

This method has two variants and tests if a string starts with the specified prefix beginning a specified index or by default at the beginning.

## Syntax:

Here is the syntax of this method:

```
public boolean startsWith(String prefix,int toffset)

or

public boolean startsWith(String prefix)
```

## Parameters:

Here is the detail of parameters:

- **prefix** -- the prefix to be matched.
- **toffset** -- where to begin looking in the string.

- **toffset** -- where to begin looking in the string.

## Return Value:

- It returns true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise.

## Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.startsWith("Welcome"));

System.out.print("Return Value :");
System.out.println(Str.startsWith("Tutorials"));

System.out.print("Return Value :");
System.out.println(Str.startsWith("Tutorials",11));
}
}
```

This produces the following result:

```
Return Value :true
Return Value :false
Return Value :true
```



# CharSequence subSequence(int beginIndex, int endIndex)

## Description:

This method returns a new character sequence that is a subsequence of this sequence.

## Syntax:

Here is the syntax of this method:

```
public CharSequence subSequence(int beginIndex,int endIndex)
```

## Parameters:

Here is the detail of parameters:

- **beginIndex** -- the begin index, inclusive.
- **endIndex** -- the end index, exclusive.

## Return Value:

```
System.out.println(Str.toCharArray());
}
}
```

This produces the following result:

```
Return Value :Welcome to Tutorialspoint.com
```

# String toLowerCase()

## Description:

This method has two variants. First variant converts all of the characters in this String to lower case using the rules of the given Locale. This is equivalent to calling toLowerCase(Locale.getDefault()).

Second variant takes locale as an argument to be used while converting into lower case.

## Syntax:

Here is the syntax of this method:

```
public String toLowerCase()

or

public String toLowerCase(Locale locale)
```



## Parameters:

Here is the detail of parameters:

- **NA**

## Return Value:

- It returns the String, converted to lowercase.

## Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.toLowerCase());
}
}
```

This produces the following result:

- It returns a copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.

## Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("   Welcome to Tutorialspoint.com   ");

System.out.print("Return Value :");
System.out.println(Str.trim());
}
}
```

This produces the following result:

```
Return Value :Welcome to Tutorialspoint.com
```

# static String valueOf(primitive data type x)

## Description:



This method has followings variants, which depend on the passed parameters. This method returns the string representation of the passed argument.

- **valueOf(boolean b):** Returns the string representation of the boolean argument.
- **valueOf(char c):** Returns the string representation of the char argument.
- **valueOf(char[] data):** Returns the string representation of the char array argument.
- **valueOf(char[] data, int offset, int count):** Returns the string representation of a specific subarray of the char array argument.
- **valueOf(double d):** Returns the string representation of the double argument.
- **valueOf(float f):** Returns the string representation of the float argument.
- **valueOf(int i):** Returns the string representation of the int argument.
- **valueOf(long l):** Returns the string representation of the long argument.
- **valueOf(Object obj):** Returns the string representation of the Object argument.

## Syntax:

Here is the syntax of this method:

```
static String valueOf(boolean b)

or

static String valueOf(char c)

or

static String valueOf(char[] data)

or
```

```
}
// 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);
}
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
Totalis11.7
Maxis3.5
```

# The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.



## Example:

The following code displays all the elements in the array myList:

```
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(double element: myList){
System.out.println(element);
}
}
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
```

# Passing Arrays to Methods:

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

```
public static void printArray(int[] array){
```

CHAPTER

15

# Java Date and Time

Java provides the **Date** class available in **java.util** package, this class encapsulates the current date and time.

The Date class supports two constructors. The first constructor initializes the object with the current date and time.

```
Date()
```

The following constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970

```
Date(long millisec)
```

Once you have a Date object available, you can call any of the following support methods to play with dates:

| SN | Methods with Description |
|---|---|
| 1 | **boolean after(Date date)**<br>Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false. |
| 2 | **boolean before(Date date)**<br>Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false. |
| 3 | **Object clone( )**<br>Duplicates the invoking Date object. |
| 4 | **int compareTo(Date date)**<br>Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date. |
| 5 | **int compareTo(Object obj)**<br>Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException. |
| 6 | **boolean equals(Object date)**<br>Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false. |
| 7 | **long getTime( )**<br>Returns the number of milliseconds that have elapsed since January 1, 1970. |
| 8 | **int hashCode( )** |

```
System.out.println("Current Date: "+ ft.format(dNow));
}
}
```

This would produce the following result:

```
CurrentDate:Sun2004.07.18 at 04:14:09 PM PDT
```

# Simple DateFormat format codes:

To specify the time format, use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

| Character | Description | Example |
|---|---|---|
| G | Era designator | AD |
| Y | Year in four digits | 2001 |
| M | Month in year | July or 07 |
| D | Day in month | 10 |
| H | Hour in A.M./P.M. (1~12) | 12 |
| H | Hour in day (0~23) | 22 |
| M | Minute in hour | 30 |
| S | Second in minute | 55 |
| S | Millisecond | 234 |
| E | Day in week | Tuesday |
| D | Day in year | 360 |
| F | Day of week in month | 2 (second Wed. in July) |
| W | Week in year | 40 |
| W | Week in month | 1 |
| A | A.M./P.M. marker | PM |
| K | Hour in day (1~24) | 24 |
| K | Hour in A.M./P.M. (0~11) | 10 |
| Z | Time zone | Eastern Standard Time |
| ' | Escape for text | Delimiter |
| " | Single quote | ` |



# Date Formatting using printf:

Date and time formatting can be done very easily using **printf** method. You use a two-letter format, starting with **t** and ending in one of the letters of the table given below. For example:

```
import java.util.Date;
```

| | | |
|---|---|---|
| Q | Milliseconds since 1970-01-01 00:00:00 GMT | 1078884319047 |

There are other useful classes related to Date and time. For more details, you can refer to Java Standard documentation.

# Parsing Strings into Dates:

The SimpleDateFormat class has some additional methods, notably parse( ) , which tries to parse a string according to the format stored in the given SimpleDateFormat object. For example:

```
import java.util.*;
import java.text.*;

public class DateDemo{

public static void main(String args[]){
SimpleDateFormat ft =new SimpleDateFormat("yyyy-MM-dd");

String input = args.length ==0?"1818-11-11": args[0];

System.out.print(input +" Parses as ");

Date t;

try{
        t = ft.parse(input);
System.out.println(t);
}catch(ParseException e){
System.out.println("Unparseable using "+ ft);
}
}
}
```



A sample run of the above program would produce the following result:

```
$ java DateDemo
1818-11-11ParsesasWedNov1100:00:00 GMT 1818
$ java DateDemo2007-12-01
2007-12-01ParsesasSatDec0100:00:00 GMT 2007
```

# Sleeping for a While:

You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, following program would sleep for 10 seconds:

```
import java.util.*;

public class SleepDemo{
public static void main(String args[]){
try{
System.out.println(new Date()+"\n");
Thread.sleep(5*60*10);
System.out.println(new Date()+"\n");
}catch(Exception e){
System.out.println("Got an exception!");
}
}
}
```

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by groupCount.

## Example:

Following example illustrates how to find a digit string from the given alphanumeric string:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
public static void main(String args[]){

// String to be scanned to find the pattern.
String line ="This order was places for QT3000! OK?";
String pattern ="(.*)(\\d+)(.*)";

// Create a Pattern object
Pattern r =Pattern.compile(pattern);

// Now create matcher object.
Matcher m = r.matcher(line);
if(m.find()){
System.out.println("Found value: "+ m.group(0));
System.out.println("Found value: "+ m.group(1));
System.out.println("Found value: "+ m.group(2));
}else{
System.out.println("NO MATCH");
}
}
}
```



This would produce the following result:

```
Found value:This order was places for QT3000! OK?
Found value:This order was places for QT300
Found value:0
```

# Regular Expression Syntax:

Here is the table listing down all the regular expression metacharacter syntax available in Java:

| Subexpression | Matches |
|---|---|
| ^ | Matches beginning of line. |
| $ | Matches end of line. |
| . | Matches any single character except newline. Using m option allows it to match newline as well. |
| [...] | Matches any single character in brackets. |
| [^...] | Matches any single character not in brackets |
| \A | Beginning of entire string |
| \z | End of entire string |
| \Z | End of entire string except allowable final line terminator. |

| | |
|---|---|
| re* | Matches 0 or more occurrences of preceding expression. |
| re+ | Matches 1 or more of the previous thing |
| re? | Matches 0 or 1 occurrence of preceding expression. |
| re{ n} | Matches exactly n number of occurrences of preceding expression. |
| re{ n,} | Matches n or more occurrences of preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of preceding expression. |
| a\| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers matched text. |
| (?: re) | Groups regular expressions without remembering matched text. |
| (?> re) | Matches independent pattern without backtracking. |
| \w | Matches word characters. |
| \W | Matches nonword characters. |
| \s | Matches whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches nonwhitespace. |
| \d | Matches digits. Equivalent to [0-9]. |
| \D | Matches nondigits. |
| \A | Matches beginning of string. |
| \Z | Matches end of string. If a newline exists, it matches just before newline. |
| \z | Matches end of string. |
| \G | Matches point where last match finished. |
| \n | Back-reference to capture group number "n" |
| \b | Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets. |
| \B | Matches nonword boundaries. |
| \n, \t, etc. | Matches newlines, carriage returns, tabs, etc. |
| \Q | Escape (quote) all characters up to \E |
| \E | Ends quoting begun with \Q |



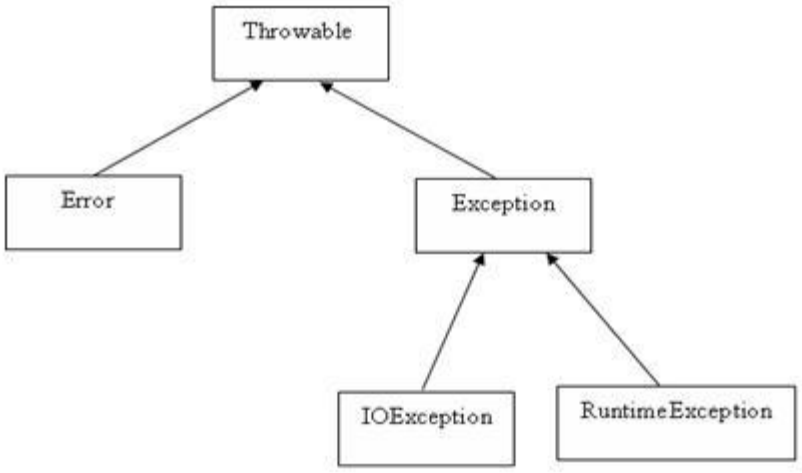
# Methods of the Matcher Class:

Here is a list of useful instance methods:

# Index Methods:

Index methods provide useful index values that show precisely where the match was found in the input string:

| SN | Methods with Description |
|---|---|




Here is a list of most common checked and unchecked **Java's Built-in Exceptions.**

## Java's Built-in Exceptions

Java defines several exception classes inside the standard package **java.lang**.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.



| Exception | Description |
|---|---|
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |

CHAPTER

20

# Java Inheritance

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance, the information is made manageable in a hierarchical order.

When we talk about inheritance, the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.



## IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

Now, based on the above example, In Object Oriented terms the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal

```
Mammal m =new Mammal();
Dog d =new Dog();

System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);
}
}
```

This would produce the following result:

```
true
true
true
```

# HAS-A relationship:

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A**certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets us look into an example:

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
privateS peed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class which makes it possible to reuse the Speed class in multiple applications.

In Object Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

```
public class extendsAnimal,Mammal{}
```

However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.

- The access level cannot be more restrictive than the overridden method's access level. For example, if the superclass method is declared public, then the overriding method in the subclass cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.



## Using the super keyword:

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

```
class Animal{

public void move(){
System.out.println("Animals can move");
}
}

class Dog extends Animal{

public void move(){
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
}
}

public class TestDog{

public static void main(String args[]){

Animal b =new Dog();// Animal reference but Dog object
   b.move();//Runs the method in Dog class
}
}
```

This would produce the following result:

```
Animals can move
Dogs can walk and run
```

CHAPTER

26

# Java Packages

Packages are used in Java inorder to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerationsss and annotations easier, etc.

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are:

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.



Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classed.

## Creating a package:

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.
The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

## Example:

Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals;

interface Animal{
public void eat();
public void travel();
```

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

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

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

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

# The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.

- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:



```
// File Name : Car.java

package vehicle;

public class Car{
// Class implementation.
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:

```
....\vehicle\Car.java
```

Now, the qualified class name and pathname would be as below:

- Class name -> vehicle.Car

- Path name -> vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

```
....\com\apple\computers\Dell.java
```

```
Vector(int size,int incr)
```

The fourth form creates a vector that contains the elements of collection c:

```
Vector(Collection c)
```

Apart from the methods inherited from its parent classes, Vector defines the following methods:

| SN | Methods with Description |
|---|---|
| 1 | **void add(int index, Object element)**<br>Inserts the specified element at the specified position in this Vector. |
| 2 | **boolean add(Object o)**<br>Appends the specified element to the end of this Vector. |
| 3 | **boolean addAll(Collection c)**<br>Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. |
| 4 | **boolean addAll(int index, Collection c)**<br>Inserts all of the elements in in the specified Collection into this Vector at the specified position. |
| 5 | **void addElement(Object obj)**<br>Adds the specified component to the end of this vector, increasing its size by one. |
| 6 | **int capacity()**<br>Returns the current capacity of this vector. |
| 7 | **void clear()**<br>Removes all of the elements from this Vector. |
| 8 | **Object clone()**<br>Returns a clone of this vector. |
| 9 | **boolean contains(Object elem)**<br>Tests if the specified object is a component in this vector. |
| 10 | **boolean containsAll(Collection c)**<br>Returns true if this Vector contains all of the elements in the specified Collection. |
| 11 | **void copyInto(Object[] anArray)**<br>Copies the components of this vector into the specified array. |
| 12 | **Object elementAt(int index)**<br>Returns the component at the specified index. |
| 13 | **Enumeration elements()**<br>Returns an enumeration of the components of this vector. |
| 14 | **void ensureCapacity(int minCapacity)**<br>Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument. |
| 15 | **boolean equals(Object o)**<br>Compares the specified Object with this Vector for equality. |
| 16 | **Object firstElement()**<br>Returns the first component (the item at index 0) of this vector. |
| 17 | **Object get(int index)**<br>Returns the element at the specified position in this Vector. |

## How to use an Iterator?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

| SN | Iterator Methods with Description |
|---|---|
| 1 | Using Java Iterator<br>Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces. |

# Using Java Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.



In general, to use an iterator to cycle through the contents of a collection, follow these steps:

- Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
- Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
- Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling ListIterator.

## The Methods Declared by Iterator:

| SN | Methods with Description |
|---|---|
| 1 | **boolean hasNext( )**<br>Returns true if there are more elements. Otherwise, returns false. |
| 2 | **Object next( )**<br>Returns the next element. Throws NoSuchElementException if there is not a next element. |
| 3 | **void remove( )**<br>Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ). |

```
      }
      System.out.println();

          // Modify objects being iterated
      ListIterator litr = al.listIterator();
      while(litr.hasNext()) {
         Object element = litr.next();
         litr.set(element + "+");
      }
      System.out.print("Modified contents of al: ");
      itr = al.iterator();
      while(itr.hasNext()) {
         Object element = itr.next();
         System.out.print(element + " ");
      }
      System.out.println();

      // Now, display the list backwards
      System.out.print("Modified list backwards: ");
      while(litr.hasPrevious()) {
         Object element = litr.previous();
         System.out.print(element + " ");
       }
       System.out.println();
   }
}
```

This would produce the following result:

```
Original contents of al: C+ A+ E+ B+ D+ F+
Modified contents of al: C++ A++ E++ B++ D++ F++
Modified list backwards: F++ D++ B++ E++ A++ C++
```



## How to use a Comparator?

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also, this interface can be used to sort any instances of any class(even classes we cannot modify).

| SN | Iterator Methods with Description |
|---|---|
| 1 | Using Java Comparator<br>Here is a list of all the methods with examples provided by Comparator Interface. |

# Using Java Comparator

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

The Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order:

## The compare Method:

```
int compare(Object obj1, Object obj2)
```

```
// Display array elements
for( E element : inputArray ){
System.out.printf("%s ", element );
}
System.out.println();
}

public static void main(String args[])
{
// Create arrays of Integer, Double and Character
Integer[] intArray ={1,2,3,4,5};
Double[] doubleArray ={1.1,2.2,3.3,4.4};
Character[] charArray ={'H','E','L','L','O'};

System.out.println("Array integerArray contains:");
  printArray( intArray  );// pass an Integer array

System.out.println("\nArray doubleArray contains:");
  printArray( doubleArray );// pass a Double array

System.out.println("\nArray characterArray contains:");
  printArray( charArray );// pass a Character array
}
}
```

This would produce the following result:

```
Array integerArray contains:
123456

Array doubleArray contains:
1.12.23.34.4

Array characterArray contains:
H E L L O
```



# Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

## Example:

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

```
public class MaximumTest
{
// determines the largest of three Comparable objects
publicstatic<T extendsComparable<T>> T maximum(T x, T y, T z)
{
T max = x;// assume x is initially the largest
if( y.compareTo( max )>0){
      max = y;// y is the largest so far
}
```

```
if( z.compareTo( max )>0){
max = z;// z is the largest now
}
return max;// returns the largest object
}
public static void main(String args[])
{
System.out.printf("Max of %d, %d and %d is %d\n\n",3,4,5, maximum(3,4,5));

System.out.printf("Maxm of %.1f,%.1f and %.1f is %.1f\n\n",6.6,8.8,7.7,
maximum(6.6,8.8,7.7));

System.out.printf("Max of %s, %s and %s is %s\n","pear",
"apple","orange", maximum("pear","apple","orange"));
}
}
```

This would produce the following result:

```
Maximum of 3,4and5is5

Maximum of 6.6,8.8and7.7is8.8

Maximum of pear, apple and orange is pear
```



# Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

# Example:

Following example illustrates how we can define a generic class:

```
public class Box<T>{

private T t;

publicvoid add(T t){
this.t = t;
}

public T get(){
return t;
}

public static void main(String[] args){
Box<Integer> integerBox =new Box<Integer>();
Box<String> stringBox =new Box<String>();

integerBox.add(newInteger(10));
stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());
System.out.printf("String Value :%s\n", stringBox.get());
```

CHAPTER

31

# Java Networking

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP:** UDP stands for User Datagram Protocol, a connectionless protocol that allows for packets of data to be transmitted between applications.

This tutorial gives good understanding on the following two subjects:

- **Socket Programming**: This is most widely used concept in Networking and it has been explained in very detail.
- **URL Processing**: This would be covered separately. Click here to learn about URL Processing in Java language.



## Url Processing

URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.

This section shows you how to write Java programs that communicate with a URL. A URL can be broken down into parts, as follows:

```
protocol://host:port/path?query#ref
```

Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority.

The following is a URL to a Web page whose protocol is HTTP:

```
http://www.amrood.com/index.htm?language=en#j2se
```

Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

```
import javax.mail.internet.*;
import javax.activation.*;

public class SendHTMLEmail
{
public static void main(String[] args)
{

// Recipient's email ID needs to be mentioned.
String to ="abcd@gmail.com";

// Sender's email ID needs to be mentioned
Stringfrom="web@gmail.com";

// Assuming you are sending email from localhost
String host ="localhost";

// Get system properties
Properties properties =System.getProperties();

// Setup mail server
properties.setProperty("mail.smtp.host", host);

// Get the default Session object.
Session session =Session.getDefaultInstance(properties);

try{
// Create a default MimeMessage object.
MimeMessage message =new MimeMessage(session);

// Set From: header field of the header.
message.setFrom(new InternetAddress(from));

// Set To: header field of the header.
message.addRecipient(Message.RecipientType.TO,
newInternetAddress(to));

// Set Subject: header field
message.setSubject("This is the Subject Line!");

// Send the actual HTML message, as big as you like
message.setContent("<h1>This is actual message</h1>",
"text/html");

// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
}catch(MessagingException mex){
    mex.printStackTrace();
}
}
}
```



Compile and run this program to send an HTML e-mail:

```
$ java SendHTMLEmail
Sent message successfully....
```

```
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

# Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

## STEP 1

You will need to override **run( )** method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of **run()** method:

```
public void run( )
```

## STEP 2

Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of **start()** method:

```
void start( );
```



# Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
```

| | |
|---|---|
| | The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | **public void interrupt()**<br>Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | **public final boolean isAlive()**<br>Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

| SN | Methods with Description |
|---|---|
| 1 | **public static void yield()**<br>Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled. |
| 2 | **public static void sleep(long millisec)**<br>Causes the currently running thread to block for at least the specified number of milliseconds. |
| 3 | **public static boolean holdsLock(Object x)**<br>Returns true if the current thread holds the lock on the given Object. |
| 4 | **public static Thread currentThread()**<br>Returns a reference to the currently running thread, which is the thread that invokes this method. |
| 5 | **public static void dumpStack()**<br>Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |



## Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
```

This would produce the following result. You can try this example again and again and you would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.......
```

# Major Java Multithreading Concepts:

While doing Multithreading programming in Java, you would need to have the following concepts very handy:

- What is thread synchronization?
- Handling threads inter communication
- Handling thread deadlock
- Major thread operations



# What is Thread synchronization?

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

## Multithreading example without Synchronization:

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces different result based on CPU availability to a thread.

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo( String name,  PrintDemo pd){
       threadName = name;
        PD = pd;
   }
   public void run() {
     PD.printCount();
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```



This produces different result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

## Multithreading example with Synchronization:

Here is the same example which prints counter value in sequence and every time we run it, it produces same result.

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}

class ThreadDemo extends Thread{
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo( String name,  PrintDemo pd){
       threadName = name;
       PD = pd;
   }
   public void run() {
     synchronized(PD) {
        PD.printCount();
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}
```

CHAPTER

34

# Java Applet Basics

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.



## Life Cycle of an Applet:

Four methods in the Applet class give you the framework on which you build any serious applet:

- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.