



Java

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the 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 of Java.

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

Audience

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

Prerequisites

Before you start practicing various types of examples given in this reference, we assume that you are already aware about computer programs and computer programming languages.

Execute Java Online

For most of the examples given in this tutorial, you will find a 'Try it' option, which you can use to execute your Java programs at the spot and enjoy your learning.

Try following the example using the 'Try it' option available at the top right corner of the following sample code box –

```
public class MyFirstJavaProgram {  
  
    public static void main(String []args) {  
        System.out.println("Hello World");  
    }  
}
```

Table of Contents

About the Tutorial	i
Audience.....	i
Prerequisites.....	i
Execute Java Online.....	i
Table of Contents	ii
 JAVA – BASICS.....	 1
1. Java – Overview	2
History of Java	3
Tools You Will Need.....	3
Try It Option	4
What is Next?	4
2. Java - Environment Setup	5
Try it Option Online	5
Local Environment Setup.....	5
Popular Java Editors	6
What is Next?	6
3. Java – Basic Syntax.....	7
First Java Program	7
Basic Syntax.....	8
Java Identifiers.....	9
Java Modifiers.....	9
Java Variables	9
Java Arrays.....	9
Java Enums	10
Java Keywords	10
Comments in Java.....	11
Using Blank Lines	12
Inheritance	12
Interfaces.....	12
What is Next?	12
4. Java – Objects & Classes.....	13
Objects in Java	13
Classes in Java.....	14
Constructors	14
How to Use Singleton Class?	15
Creating an Object.....	17
Accessing Instance Variables and Methods.....	18
Source File Declaration Rules	20
Java Package.....	20
Import Statements	21
A Simple Case Study	21
What is Next?	23

5. Java – Basic Datatypes	24
Primitive Datatypes	24
Reference Datatypes	26
Java Literals	26
What is Next?	28
6. Java – Variable Types	29
Local Variables	29
Instance Variables	31
Class/static Variables.....	33
What is Next?	34
7. Java – Modifier Types	35
Java Access Modifiers	35
Java Non-Access Modifiers	38
The Static Modifier	38
The Final Modifier	39
The Abstract Modifier.....	41
Access Control Modifiers	43
Non-Access Modifiers.....	44
What is Next?	44
8. Java – Basic Operators	45
The Arithmetic Operators.....	45
The Relational Operators.....	47
The Bitwise Operators	49
The Logical Operators.....	52
The Assignment Operators	53
Miscellaneous Operators.....	57
Precedence of Java Operators.....	59
What is Next?	59
9. Java – Loop Control.....	60
While Loop in Java	61
for Loop in Java.....	62
Do While Loop in Java	65
Loop Control Statements.....	67
Break Statement in Java	67
Continue Statement in Java	69
Enhanced for loop in Java.....	70
What is Next?	71
10. Java – Decision Making	72
If Statement in Java	73
If-else Statement in Java.....	74
The if...else if...else Statement	76
Nested if Statement in Java	77
Switch Statement in Java	78
The ? : Operator:	80
What is Next?	81

11. Java – Numbers Class	82
Number Methods	83
Java XXXValue Method	86
Java – compareTo() Method	87
Java – equals() Method	88
Java – valueOf() Method	89
Java – toString() Method	91
Java – parseInt() Method	92
Java – abs() Method	93
Java – ceil() Method	94
Java – floor() Method	95
Java – rint() Method	96
Java – round() Method	97
Java – min() Method	98
Java – max() Method	99
Java – exp() Method	100
Java – log() Method	101
Java – pow() Method	102
Java – sqrt() Method	103
Java – sin() Method	104
Java – cos() Method	105
Java – tan() Method	106
Java – asin() Method	107
Java – acos() Method	108
Java – atan() Method	109
Java – atan2() Method	110
Java – toDegrees() Method	111
Java – toRadians() Method	112
Java – random() Method	113
What is Next?	114
12. Java – Character Class	115
Escape Sequences	115
Character Methods	117
Java – isLetter() Method	117
Java – isDigit() Method	118
Java – isWhitespace() Method	119
Java – isUpperCase() Method	120
Java – isLowerCase() Method	121
Java – toUpperCase() Method	122
Java – toLowerCase() Method	123
Java – toString() Method	124
What is Next?	125
13. Java – Strings Class	126
Creating Strings	126
Java – String Buffer & String Builder Classes	126
StringBuffer Methods	127
Java – String Buffer append() Method	128
Java – String Buffer reverse() Method	129

Java – String Buffer delete() Method	130
Java – String Buffer insert() Method	131
Java – String Buffer replace() Method	132
String Length.....	135
Concatenating Strings.....	136
Creating Format Strings	136
String Methods	137
Java – String charAt() Method.....	142
Java – String compareTo(Object o) Method.....	143
Java – String compareTo(String anotherString) Method	144
Java – String compareToIgnoreCase() Method	145
Java – String concat() Method.....	146
Java – String contentEquals() Method.....	147
Java – String copyValueOf(char[] data) Method	148
Java – String copyValueOf(char[] data, int offset, int count) Method.....	149
Java – String endsWith() Method	150
Java – String equals() Method	151
Java – String equalsIgnoreCase() Method	152
Java – String getBytes(String charsetName) Method.....	154
Java – String getBytes() Method.....	155
Java – String getChars() Method	156
Java – String hashCode() Method.....	157
Java – String indexOf(int ch) Method	158
Java – String indexOf(int ch, int fromIndex) Method	159
Java – String indexOf(String str) Method	160
Java – String indexOf(String str, int fromIndex) Method.....	161
Java – String Intern() Method.....	162
Java – String lastIndexOf(int ch) Method	163
Java – String lastIndexOf(int ch, int fromIndex) Method	164
Java – String lastIndexOf(String str) Method.....	165
Java – String lastIndexOf(String str, int fromIndex) Method	166
Java – String length() Method	167
Java – String matches() Method	168
Java – String regionMatches() Method	169
Java – String regionMatches() Method	171
Java – String replace() Method.....	173
Java – String replaceAll() Method.....	174
Java – String replaceFirst() Method.....	175
Java – String split() Method	176
Java – String split() Method	178
Java – String startsWith() Method.....	180
Java – String startsWith() Method.....	181
Java – String subsequence() Method	182
Java – String substring() Method.....	183
Java – String substring() Method.....	184
Java – String toCharArray() Method	186
Java – String toLowerCase() Method.....	187
Java – String toLowerCase() Method.....	188
Java – String toString() Method.....	189
Java – String toUpperCase() Method.....	189

Java – String toUpperCase() Method.....	190
Java – String trim() Method.....	191
Java – String valueOf() Method.....	192
14. Java – Arrays.....	196
Declaring Array Variables.....	196
Creating Arrays.....	196
Processing Arrays.....	198
The foreach Loops.....	199
Passing Arrays to Methods.....	199
Returning an Array from a Method.....	200
The Arrays Class.....	200
15. Java – Date & Time.....	202
Getting Current Date & Time.....	203
Date Comparison.....	204
Simple DateFormat Format Codes.....	205
Date and Time Conversion Characters.....	208
Parsing Strings into Dates.....	209
Sleeping for a While.....	210
Measuring Elapsed Time.....	211
GregorianCalendar Class.....	212
16. Java – Regular Expressions.....	218
Capturing Groups.....	218
Regular Expression Syntax.....	220
Methods of the Matcher Class.....	223
17. Java – Methods.....	230
Creating Method.....	230
Method Calling.....	231
The void Keyword.....	232
Passing Parameters by Value.....	233
Method Overloading.....	235
Using Command-Line Arguments.....	236
The Constructors.....	237
Parameterized Constructor.....	238
The this keyword.....	239
Variable Arguments(var-args).....	242
The finalize() Method.....	243
18. Java – Files and I/O.....	244
Stream.....	244
Standard Streams.....	247
Reading and Writing Files.....	248
ByteArrayInputStream.....	250
DataInputStream.....	253
FileOutputStream.....	255
ByteArrayOutputStream.....	256
DataOutputStream.....	259
File Navigation and I/O.....	261

File Class	262
Directories in Java.....	272
Listing Directories	273
19. Java – Exceptions	274
Exception Hierarchy.....	275
Built-in Exceptions	276
Exceptions Methods	278
Catching Exceptions.....	279
Multiple Catch Blocks	280
Catching Multiple Type of Exceptions	281
The Throws/Throw Keywords	281
The Finally Block	282
The try-with-resources	284
User-defined Exceptions.....	286
Common Exceptions.....	289
20. Java – Inner Classes.....	290
Nested Classes	290
Inner Classes (Non-static Nested Classes)	291
Accessing the Private Members	292
Method-local Inner Class	293
Anonymous Inner Class	294
Anonymous Inner Class as Argument.....	295
Static Nested Class.....	296
JAVA - OBJECT ORIENTED.....	299
21. Java – Inheritance	300
extends Keyword	300
Sample Code.....	300
The super keyword	302
Invoking Superclass Constructor	305
IS-A Relationship.....	306
The instanceof Keyword	308
HAS-A relationship.....	309
Types of Inheritance	309
22. Java – Overriding	311
Rules for Method Overriding.....	313
Using the super Keyword	314
23. Java – Polymorphism	315
Virtual Methods.....	316
24. Java – Abstraction.....	320
Abstract Class	320
Inheriting the Abstract Class.....	323
Abstract Methods.....	324

25. Java – Encapsulation	326
Benefits of Encapsulation	328
26. Java – Interfaces	329
Declaring Interfaces.....	330
Implementing Interfaces	330
Extending Interfaces.....	332
Extending Multiple Interfaces	333
Tagging Interfaces	333
27. Java – Packages.....	334
Creating a Package	334
The import Keyword	336
The Directory Structure of Packages	337
Set CLASSPATH System Variable.....	339
JAVA – ADVANCED	340
28. Java – Data Structures.....	341
The Enumeration	341
The BitSet	343
The Vector	348
The Stack	355
The Dictionary	358
The Hashtable.....	362
The Properties	366
29. Java – Collections Framework	370
The Collection Interfaces	371
The Collection Interface	372
The List Interface	375
The Set Interface	378
The SortedSet Interface.....	380
The Map Interface	382
The Map.Entry Interface.....	384
The SortedMap Interface	386
The Enumeration Interface.....	388
The Collection Classes	389
The LinkedList Class.....	391
The ArrayList Class.....	395
The HashSet Class.....	399
The LinkedHashSet Class	402
The TreeSet Class.....	403
The HashMap Class.....	406
The TreeMap Class	409
The WeakHashMap Class	412
The LinkedHashMap Class	415
The IdentityHashMap Class	418
The Vector Class	422
The Stack Class	428

The Dictionary Class	430
The Map Interface	431
The Hashtable Class.....	433
The Properties Class	437
The BitSet Class	440
The Collection Algorithms	444
How to Use an Iterator ?	449
How to Use a Comparator ?	453
Summary	455
30. Java – Generics	456
Generic Methods	456
Bounded Type Parameters	458
Generic Classes	459
31. Java – Serialization.....	461
Serializing an Object	462
Deserializing an Object	463
32. Java – Networking.....	465
URL Processing	465
URL Class Methods	466
URLConnections Class Methods	469
Socket Programming	472
ServerSocket Class Methods	473
Socket Class Methods.....	474
InetAddress Class Methods	476
Socket Client Example	476
Socket Server Example	478
33. Java – Sending E-mail	480
Send a Simple E-mail	480
Send an HTML E-mail.....	482
Send Attachment in E-mail	484
User Authentication Part	486
34. Java – Multithreading	487
Life Cycle of a Thread	487
Thread Priorities	488
Create a Thread by Implementing a Runnable Interface	488
Create a Thread by Extending a Thread Class	490
Thread Methods	493
Major Java Multithreading Concepts	498
Thread Synchronization.....	498
Interthread Communication	503
Thread Deadlock.....	506
Thread Control.....	509
35. Java – Applet Basics	514
Life Cycle of an Applet	514
A "Hello, World" Applet.....	515

The Applet Class	515
Invoking an Applet.....	516
HTML <applet> Tag.....	516
HTML Attribute Reference	518
HTML Events Reference	520
Getting Applet Parameters.....	525
Specifying Applet Parameters	526
Application Conversion to Applets	526
Event Handling	527
Displaying Images	529
Playing Audio.....	531
36. Java – Documentation Comments.....	533
What is Javadoc?	533
The javadoc Tags	534

Java – Basics

1. Java – Overview

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

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

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

Java is:

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

- **Distributed:** Java is designed for the distributed environment of the internet.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java

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

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

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

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

Tools You Will Need

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You will also need the following softwares:

- Linux 7.1 or Windows xp/7/8 operating system
- Java JDK 8
- Microsoft Notepad or any other text editor

This tutorial will provide the necessary skills to create GUI, networking, and web applications using Java.

Try It Option

We have provided you with an option to compile and execute available code online. Just click the **Try it** button available at the top-right corner of the code window to compile and execute the available code. There are certain examples which cannot be executed online, so we have skipped those examples.

```
public class MyFirstJavaProgram {  
  
    public static void main(String []args) {  
        System.out.println("Hello World");  
    }  
}
```

There may be a case that you do not see the result of the compiled/executed code. In such case, you can re-try to compile and execute the code using **execute** button available in the compilation pop-up window.

What is Next?

The next chapter will guide you to how you can obtain Java and its documentation. Finally, it instructs you on how to install Java and prepare an environment to develop Java applications.

2. Java - Environment Setup

In this chapter, we will discuss on the different aspects of setting up a congenial environment for Java.

Try it Option Online

You really do not need to set up your own environment to start learning Java programming language. Reason is very simple, we already have Java Programming environment setup online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using **Try it** option available at the top right corner of the following sample code box:

```
public class MyFirstJavaProgram {  
  
    public static void main(String []args) {  
        System.out.println("Hello World");  
    }  
}
```

For most of the examples given in this tutorial, you will find the **Try it** option, which you can use to execute your programs and enjoy your learning.

Local Environment Setup

If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Following are the steps to set up the environment.

Java SE is freely available from the link [Download Java](#). You can 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 will need to set environment variables to point to correct installation directories:

Setting Up the Path for Windows

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

- Right-click on 'My Computer' and select 'Properties'.
- Click 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 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**

Popular Java Editors

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following:

- **Notepad:** On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- **Netbeans:** A Java IDE that is open-source and free, which can be downloaded from <http://www.netbeans.org/index.html>.
- **Eclipse:** A Java IDE developed by the eclipse open-source community and can be downloaded from <http://www.eclipse.org/>.

What is Next?

Next chapter will teach you how to write and run your first Java program and some of the important basic syntaxes in Java needed for developing applications.

3. Java – Basic Syntax

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

- **Object** - Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** - A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- **Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** - Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     */  
  
    public static void main(String []args) {  
        System.out.println("Hello World"); // prints Hello World  
    }  
}
```

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

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.

- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
Hello World
```

Basic Syntax

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

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

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

Example: *class MyFirstJavaClass*

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

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

Example: *public void myMethodName()*

- **Program File Name** - Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as '*MyFirstJavaProgram.java*'

- **public static void main(String args[])** - Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers

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

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

- All identifiers should begin with a letter (A to Z or a to z), currency character (\$) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, \$salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers:

- **Access Modifiers:** default, public, protected, private
- **Non-access Modifiers:** final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

Java Variables

Following are the types of variables in Java:

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

Java Arrays

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap. We will look into how to declare, construct, and initialize in the upcoming chapters.

Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

Example

```
class FreshJuice {

    enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
    FreshJuiceSize size;
}

public class FreshJuiceTest {

    public static void main(String args[]){
        FreshJuice juice = new FreshJuice();
        juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
        System.out.println("Size: " + juice.size);
    }
}
```

The above example will produce the following result:

```
Size: MEDIUM
```

Note: Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords

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

abstract	assert	boolean	break
byte	case	catch	char

class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Comments in Java

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

```
public class MyFirstJavaProgram{

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

    public static void main(String []args){
        // This is an example of single line comment
        /* This is also an example of single line comment. */
        System.out.println("Hello World");
    }
}
```

Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

Inheritance

In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**.

Interfaces

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.

What is Next?

The next section explains about Objects and classes in Java programming. At the end of the session, you will be able to get a clear picture as to what are objects and what are classes in Java.

4. Java – Objects & Classes

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter, we will look into the concepts - Classes and Objects.

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

Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

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

Following is a sample of a class.

```
public class Dog{  
    String breed;  
    int ageC  
    String color;  
  
    void barking(){  
    }  
  
    void hungry(){  
    }  
  
    void sleeping(){  
    }  
}
```

A class can contain any of the following variable types.

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

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

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

Constructors

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

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

Following is an example of a constructor:

```
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.

Note: We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

How to Use Singleton Class?

The Singleton's purpose is to control object creation, limiting the number of objects to only one. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources, such as database connections or sockets.

For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

Implementing Singletons

Example 1

The easiest implementation consists of a private constructor and a field to hold its result, and a static accessor method with a name like getInstance().

The private field can be assigned from within a static initializer block or, more simply, using an initializer. The getInstance() method (which must be public) then simply returns this instance –

```
// File Name: Singleton.java
public class Singleton {

    private static Singleton singleton = new Singleton( );

    /* A private Constructor prevents any other
     * class from instantiating.
     */
    private Singleton(){ }

    /* Static 'instance' method */
    public static Singleton getInstance( ) {
        return singleton;
    }

    /* Other methods protected by singleton-ness */
    protected static void demoMethod( ) {
        System.out.println("demoMethod for singleton");
    }
}
```

Here is the main program file, where we will create a singleton object:

```
// File Name: SingletonDemo.java
public class SingletonDemo {
    public static void main(String[] args) {
        Singleton tmp = Singleton.getInstance( );
        tmp.demoMethod( );
    }
}
```

This will produce the following result –

```
demoMethod for singleton
```

Example 2

Following implementation shows a classic Singleton design pattern:

```
public class ClassicSingleton {  
  
    private static ClassicSingleton instance = null;  
    private ClassicSingleton() {  
        // Exists only to defeat instantiation.  
    }  
    public static ClassicSingleton getInstance() {  
        if(instance == null) {  
            instance = new ClassicSingleton();  
        }  
        return instance;  
    }  
}
```

The ClassicSingleton class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

Here, ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.

Creating an Object

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

There are three steps when creating an object from a class:

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

Following is an example of creating an object:

```
public class Puppy{

    public Puppy(String name){
        // This constructor has one parameter, name.
        System.out.println("Passed Name is :" + name );
    }

    public static void main(String []args){
        // Following statement would create an object myPuppy
        Puppy myPuppy = new Puppy( "tommy" );
    }
}
```

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

```
Passed Name is :tommy
```

Accessing Instance Variables and Methods

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

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

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

Example

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

```
public class Puppy{

    int puppyAge;

    public Puppy(String name){
        // This constructor has one parameter, name.
        System.out.println("Name chosen is :" + name );
    }

    public void setAge( int age ){
        puppyAge = age;
    }

    public int getAge( ){
        System.out.println("Puppy's age is :" + puppyAge );
        return puppyAge;
    }

    public static void main(String []args){
        /* Object creation */
        Puppy myPuppy = new Puppy( "tommy" );

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

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

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

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

```
Name chosen 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.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import Statements

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory `java_installation/java/io`:

```
import java.io.*;
```

A Simple Case Study

For our case study, we will be creating two classes. They are `Employee` and `EmployeeTest`.

First open notepad and add the following code. Remember this is the `Employee` class and the class is a public class. Now, save this source file with the name `Employee.java`.

The `Employee` class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

```
import java.io.*;

public class Employee{

    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name){
        this.name = name;
    }
    // Assign the age of the Employee to the variable age.
    public void empAge(int empAge){
        age = empAge;
    }
    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig){
        designation = empDesig;
    }
    /* Assign the salary to the variable salary.*/
```



```

    public void empSalary(double empSalary){
        salary = empSalary;
    }
    /* Print the Employee details */
    public void printEmployee(){
        System.out.println("Name:" + name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}

```

As mentioned previously in this tutorial, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

```

import java.io.*;

public class EmployeeTest{
    public static void main(String args[]){
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
        Employee empTwo = new Employee("Mary Anne");

        // Invoking methods for each object created
        empOne.empAge(26);
        empOne.empDesignation("Senior Software Engineer");
        empOne.empSalary(1000);
        empOne.printEmployee();
        empTwo.empAge(21);
        empTwo.empDesignation("Software Engineer");
        empTwo.empSalary(500);
        empTwo.printEmployee();
    } }

```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows:

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

What is Next?

In the next session, we will discuss the basic data types in Java and how they can be used when developing Java applications.

5. Java – Basic Datatypes

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

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

There are two data types available in Java:

- Primitive Datatypes
- Reference/Object Datatypes

Primitive Datatypes

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

byte:

- Byte data type is an 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 datatype is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer
- Example: byte a = 100 , byte b = -50

short:

- Short datatype 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 datatype can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0

- Example: short s = 10000, short r = -20000

int:

- Int datatype 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) ($2^{31} - 1$)
- Integer 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 datatype 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, long b = -200000L

float:

- Float datatype 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 datatype is never used for precise values such as currency
- Example: float f1 = 234.5f

double:

- double datatype is a double-precision 64-bit IEEE 754 floating point
- This datatype is generally used as the default data type for decimal values, generally the default choice
- Double datatype should never be used for precise values such as currency
- Default value is 0.0d

- Example: double d1 = 123.4

boolean:

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

char:

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

Reference Datatypes

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

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

<code>\"</code>	Double quote
<code>\'</code>	Single quote
<code>\\</code>	backslash
<code>\ddd</code>	Octal character (ddd)
<code>\uxxxx</code>	Hexadecimal UNICODE character (xxxx)

What is Next?

This chapter explained the various data types. The next topic explains different variable types and their usage. This will give you a good understanding on how they can be used in the Java classes, interfaces, etc.

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. Following is the basic form of a variable declaration:

```
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 is 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.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

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

```
public class Test{
    public void pupAge(){
        int age = 0;
        age = age + 7;
        System.out.println("Puppy age is : " + age);
    }

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

This will produce the following result:

```
Puppy age is: 7
```

Example

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test{
    public void pupAge(){
        int age;
        age = age + 7;
        System.out.println("Puppy age is : " + age);
    }

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

This will produce the following error while compiling it:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
          ^
1 error
```

Instance Variables

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

Example

```
import java.io.*;

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

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

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

    // The salary variable is assigned a value.
    public void setSalary(double empSal){
        salary = empSal;
    }

    // This method prints the employee details.
    public void printEmp(){
        System.out.println("name : " + name );
        System.out.println("salary :" + salary);
    }

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

This will produce the following result:

```
name : Ransika  
salary :1000.0
```

Class/static Variables

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

Example

```
import java.io.*;
public class Employee{
    // salary variable is a private static variable
    private static double salary;

    // DEPARTMENT is a constant
    public static final String DEPARTMENT = "Development ";
    public static void main(String args[]){
        salary = 1000;
        System.out.println(DEPARTMENT + "average salary:" + salary);
    }
}
```

This will produce the following result:

```
Development average salary:1000
```

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

What is Next?

You already have used access modifiers (public & private) in this chapter. The next chapter will explain Access Modifiers and Non-Access Modifiers in detail.

7. Java – Modifier Types

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

- [Java Access Modifiers](#)
- [Non Access Modifiers](#)

Java Access Modifiers

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

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "1.5.1";

boolean processOrder() {
    return true;
}
```

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 hides 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 needs 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.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in an interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method:

```
class AudioPlayer {
    protected boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}

class StreamingAudioPlayer {
    boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}
```

Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used *protected* modifier.

Access Control and Inheritance

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.

Java Non-Access Modifiers

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

- 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* keyword 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* keyword 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 will produce the 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 without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

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

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

The abstract method ends with a semicolon. Example: public 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 keyword 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 modifier 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*.

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following example.

```
public class className {
    // ...
}
private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
public static void main(String[] arguments) {
    // body of method
}
```

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:

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

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.

What is Next?

In the next section, we will be discussing about Basic Operators used in Java Language. The chapter will give you an overview of how these operators can be used during application development.

8. Java – Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Sr.No.	Operator and Example
1	+ (Addition) Adds values on either side of the operator Example: A + B will give 30
2	- (Subtraction) Subtracts right-hand operand from left-hand operand Example: A - B will give -10
3	* (Multiplication) Multiplies values on either side of the operator Example: A * B will give 200

4	/ (Division) Divides left-hand operand by right-hand operand Example: B / A will give 2
5	% (Modulus) Divides left-hand operand by right-hand operand and returns remainder Example: B % A will give 0
6	++ (Increment) Increases the value of operand by 1 Example: B++ gives 21
7	-- (Decrement) Decreases the value of operand by 1 Example: B-- gives 19

Example

The following program is a simple example which demonstrates the arithmetic operators. Copy and paste the following Java program in Test.java file, and compile and run this program:

```
public class Test {
    public static void main(String args[]) {
        int a = 10;
        int b = 20;
        int c = 25;
        int d = 25;
        System.out.println("a + b = " + (a + b) );
        System.out.println("a - b = " + (a - b) );
        System.out.println("a * b = " + (a * b) );
        System.out.println("b / a = " + (b / a) );
        System.out.println("b % a = " + (b % a) );
        System.out.println("c % a = " + (c % a) );
        System.out.println("a++ = " + (a++) );
        System.out.println("b-- = " + (a--) );
    }
}
```

```
// Check the difference in d++ and ++d
System.out.println("d++   = " + (d++) );
System.out.println("++d   = " + (++d) );
} }
```

This will produce the following result:

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then:

Sr.No.	Operator and Description
1	== (equal to) Checks if the values of two operands are equal or not, if yes then condition becomes true. Example: (A == B) is not true.
2	!= (not equal to) Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. Example: (A != B) is true.

3	> (greater than) Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. Example: (A > B) is not true.
4	< (less than) Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. Example: (A < B) is true.
5	>= (greater than or equal to) Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. Example: (A >= B) is not true.
6	<= (less than or equal to) 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: (A <= B) is true.

Example

The following program is a simple example that demonstrates the relational operators. Copy and paste the following Java program in Test.java file and compile and run this program.

```
public class Test {

    public static void main(String args[]) {
        int a = 10;
        int b = 20;
        System.out.println("a == b = " + (a == b) );
        System.out.println("a != b = " + (a != b) );
        System.out.println("a > b = " + (a > b) );
        System.out.println("a < b = " + (a < b) );
        System.out.println("b >= a = " + (b >= a) );
        System.out.println("b <= a = " + (b <= a) );
    }
}
```

This will produce the following result:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Sr. No.	Operator and Description
1	<p>& (bitwise and)</p> <p>Binary AND Operator copies a bit to the result if it exists in both operands.</p> <p>Example: (A & B) will give 12 which is 0000 1100</p>

2	<p> (bitwise or)</p> <p>Binary OR Operator copies a bit if it exists in either operand.</p> <p>Example: (A B) will give 61 which is 0011 1101</p>
3	<p>^ (bitwise XOR)</p> <p>Binary XOR Operator copies the bit if it is set in one operand but not both.</p> <p>Example: (A ^ B) will give 49 which is 0011 0001</p>
4	<p>~ (bitwise compliment)</p> <p>Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.</p> <p>Example: (~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.</p>
5	<p><< (left shift)</p> <p>Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.</p> <p>Example: A << 2 will give 240 which is 1111 0000</p>
6	<p>>> (right shift)</p> <p>Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.</p> <p>Example: A >> 2 will give 15 which is 1111</p>
7	<p>>>> (zero fill right shift)</p> <p>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.</p> <p>Example: A >>>2 will give 15 which is 0000 1111</p>

Example

The following program is a simple example that demonstrates the bitwise operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {  
  
    public static void main(String args[]) {  
        int a = 60;  /* 60 = 0011 1100 */  
        int b = 13;  /* 13 = 0000 1101 */  
        int c = 0;  
  
        c = a & b;      /* 12 = 0000 1100 */  
        System.out.println("a & b = " + c );  
  
        c = a | b;      /* 61 = 0011 1101 */  
        System.out.println("a | b = " + c );  
  
        c = a ^ b;      /* 49 = 0011 0001 */  
        System.out.println("a ^ b = " + c );  
  
        c = ~a;         /* -61 = 1100 0011 */  
        System.out.println("~a = " + c );  
  
        c = a << 2;      /* 240 = 1111 0000 */  
        System.out.println("a << 2 = " + c );  
  
        c = a >> 2;      /* 15 = 1111 */  
        System.out.println("a >> 2 = " + c );  
  
        c = a >>> 2;     /* 15 = 0000 1111 */  
        System.out.println("a >>> 2 = " + c );  
    }  
}
```

This will produce the following result:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

The Logical Operators

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description
1	&& (logical and) Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. Example: (A && B) is false.
2	 (logical or) Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. Example: (A B) is true.
3	! (logical not) 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: !(A && B) is true.

Example

The following simple example program demonstrates the logical operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {

    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;

        System.out.println("a && b = " + (a&&b));

        System.out.println("a || b = " + (a||b) );

        System.out.println("!(a && b) = " + !(a && b));
    }
}
```

This will produce the following result:

```
a && b = false
a || b = true
!(a && b) = true
```

The Assignment Operators

Following are the assignment operators supported by Java language:

Sr. No.	Operator and Description
1	<p>=</p> <p>Simple assignment operator. Assigns values from right side operands to left side operand.</p> <p>Example: C = A + B will assign value of A + B into C</p>

2	<p>+=</p> <p>Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.</p> <p>Example: $C += A$ is equivalent to $C = C + A$</p>
3	<p>-=</p> <p>Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.</p> <p>Example: $C -= A$ is equivalent to $C = C - A$</p>
4	<p>*=</p> <p>Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.</p> <p>Example: $C *= A$ is equivalent to $C = C * A$</p>
5	<p>/=</p> <p>Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.</p> <p>Example: $C /= A$ is equivalent to $C = C / A$</p>
6	<p>%=</p> <p>Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.</p> <p>Example: $C \% = A$ is equivalent to $C = C \% A$</p>
7	<p><<=</p> <p>Left shift AND assignment operator.</p> <p>Example: $C <<= 2$ is same as $C = C << 2$</p>
8	<p>>>=</p> <p>Right shift AND assignment operator</p> <p>Example: $C >>= 2$ is same as $C = C >> 2$</p>

9	&= Bitwise AND assignment operator. Example: C &= 2 is same as C = C & 2
10	^= bitwise exclusive OR and assignment operator. Example: C ^= 2 is same as C = C ^ 2
11	 = bitwise inclusive OR and assignment operator. Example: C = 2 is same as C = C 2

Example

The following program is a simple example that demonstrates the assignment operators. Copy and paste the following Java program in Test.java file. Compile and run this program:

```
public class Test {

    public static void main(String args[]) {

        int a = 10;
        int b = 20;
        int c = 0;

        c = a + b;
        System.out.println("c = a + b = " + c );

        c += a ;
        System.out.println("c += a = " + c );

        c -= a ;
        System.out.println("c -= a = " + c );

        c *= a ;
        System.out.println("c *= a = " + c );
    }
}
```

```

    a = 10;
    c = 15;
    c /= a ;
    System.out.println("c /= a = " + c );

    a = 10;
    c = 15;
    c %= a ;
    System.out.println("c %= a  = " + c );

    c <<= 2 ;
    System.out.println("c <<= 2 = " + c );

    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );

    c >>= 2 ;
    System.out.println("c >>= a = " + c );

    c &= a ;
    System.out.println("c &= 2  = " + c );

    c ^= a ;
    System.out.println("c ^= a   = " + c );

    c |= a ;
    System.out.println("c |= a   = " + c );
}
}

```

This will produce the following result:

```

c = a + b = 30
c += a  = 40
c -= a  = 30
c *= a  = 300

```

```

c /= a = 1
c %= a = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a = 0
c ^= a = 10
c |= a = 10

```

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is an example:

```

public class Test {

    public static void main(String args[]){
        int a, b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b );

        b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}

```

This will produce the following result:

```

Value of b is : 30
Value of b is : 20

```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as:

```
( Object reference variable ) instanceof (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example:

```
public class Test {  
  
    public static void main(String args[]){  
        String name = "James";  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

This will produce the following result:

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {}  
  
public class Car extends Vehicle {  
    public static void main(String args[]){  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println( result );  
    }  
}
```

This will produce the following result:

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ -- ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left

What is Next?

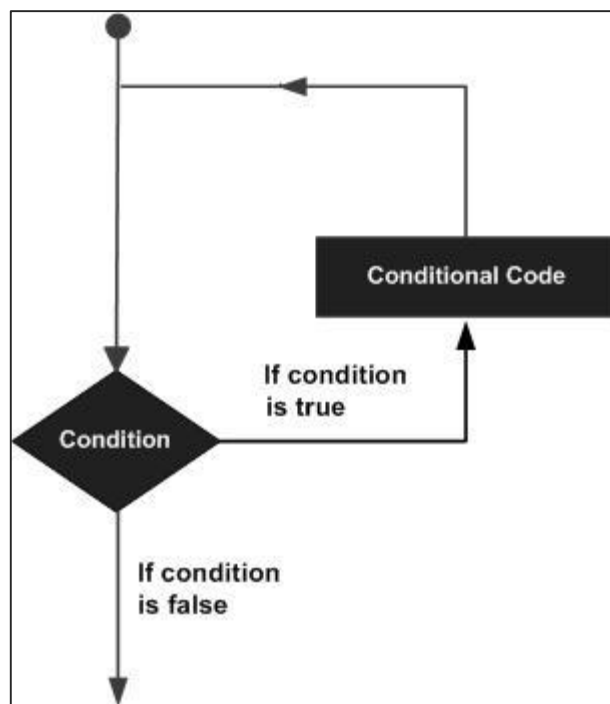
The next chapter will explain about loop control in Java programming. The chapter will describe various types of loops and how these loops can be used in Java program development and for what purposes they are being used.

9. Java – Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Java programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
<u>while loop</u>	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
<u>for loop</u>	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
<u>do...while loop</u>	Like a while statement, except that it tests the condition at the end of the loop body.

While Loop in Java

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop is:

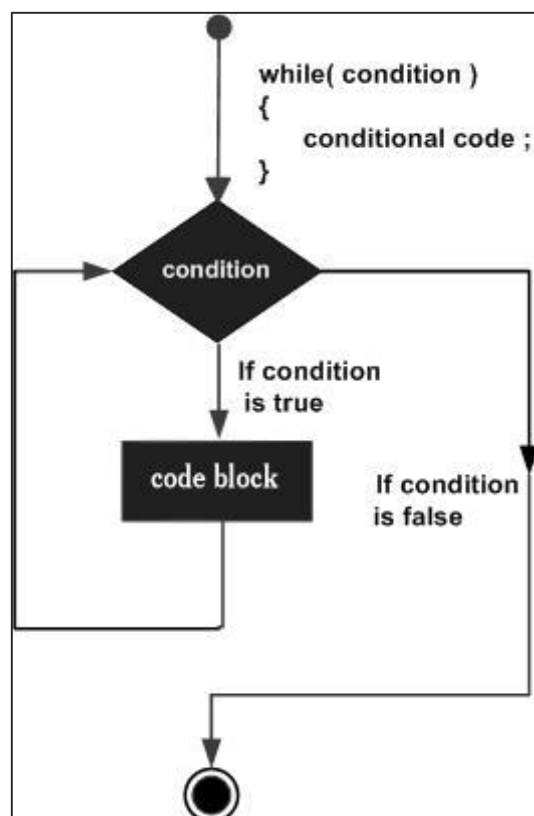
```
while(Boolean_expression)
{
    //Statements
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        int x = 10;  
  
        while( x < 20 ) {  
            System.out.print("value of x : " + x );  
            x++;  
            System.out.print("\n");  
        }  
    }  
}
```

This will produce the following result:

```
value of x : 10  
value of x : 11  
value of x : 12  
value of x : 13  
value of x : 14  
value of x : 15  
value of x : 16  
value of x : 17  
value of x : 18  
value of x : 19
```

for Loop in Java

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for** loop is useful when you know how many times a task is to be repeated.

Syntax

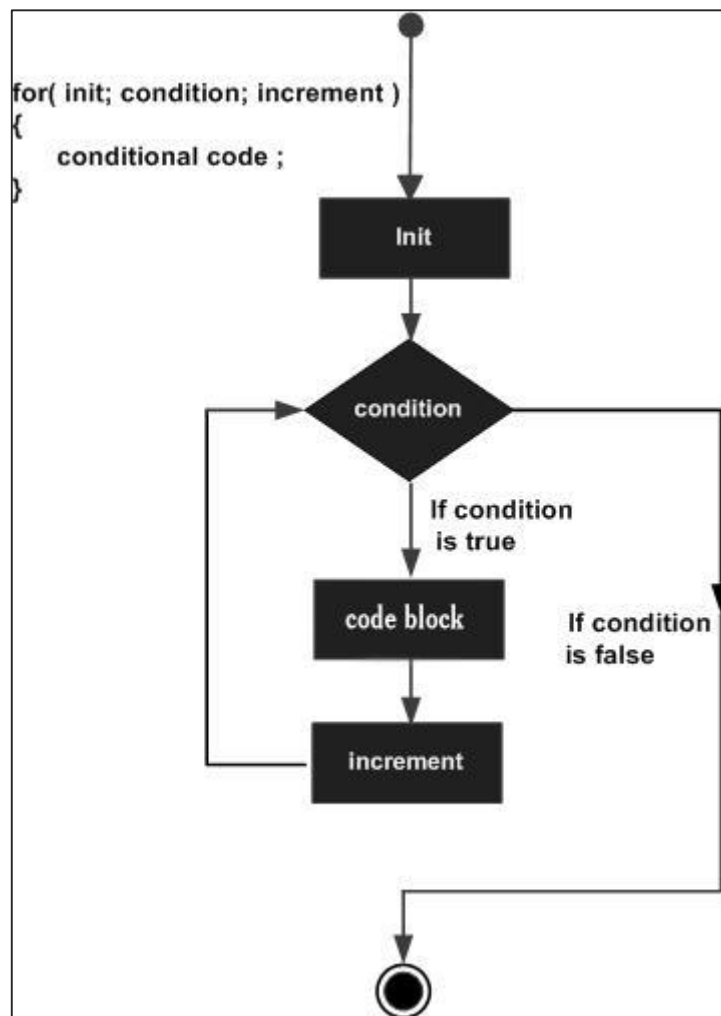
The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)
{
    //Statements
}
```

Here is the flow of control in a **for** loop:

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram



Example

Following is an example code of the for loop in Java.

```

public class Test {

    public static void main(String args[]) {

        for(int x = 10; x < 20; x = x+1) {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }
    }
}
  
```

```
}
```

This will produce the following result:

```
value of x : 10  
value of x : 11  
value of x : 12  
value of x : 13  
value of x : 14  
value of x : 15  
value of x : 16  
value of x : 17  
value of x : 18  
value of x : 19
```

Do While Loop in Java

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

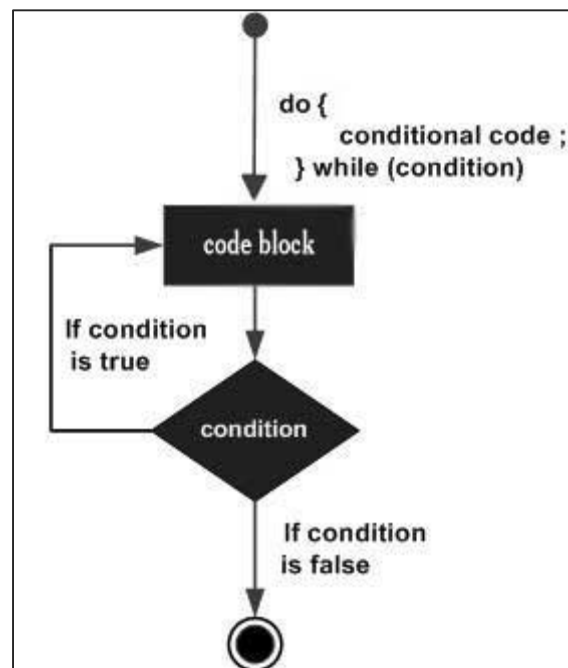
Following is the syntax of a do...while loop:

```
do  
{  
    //Statements  
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

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

Flow Diagram



Example

```
public class Test {  
  
    public static void main(String args[]){  
        int x = 10;  
  
        do{  
            System.out.print("value of x : " + x );  
            x++;  
            System.out.print("\n");  
        }while( x < 20 );  
    }  
}
```

This will produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements. Click the following links to check their detail.

Control Statement	Description
<u>break statement</u>	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
<u>continue statement</u>	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Break Statement in Java

The **break** statement in Java programming language has the following two usages:

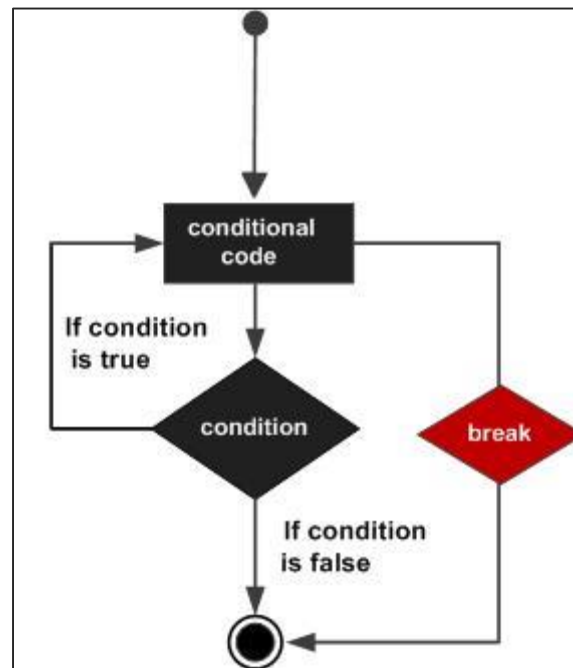
- When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

Syntax

The syntax of a break is a single statement inside any loop:

```
break;
```

Flow Diagram



Example

```
public class Test {  
  
    public static void main(String args[]) {  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ) {  
            if( x == 30 ) {  
                break;  
            }  
            System.out.print( x );  
            System.out.print("\n");  
        }  
    }  
}
```

This will produce the following result:

```
10
20
```

Continue Statement in Java

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

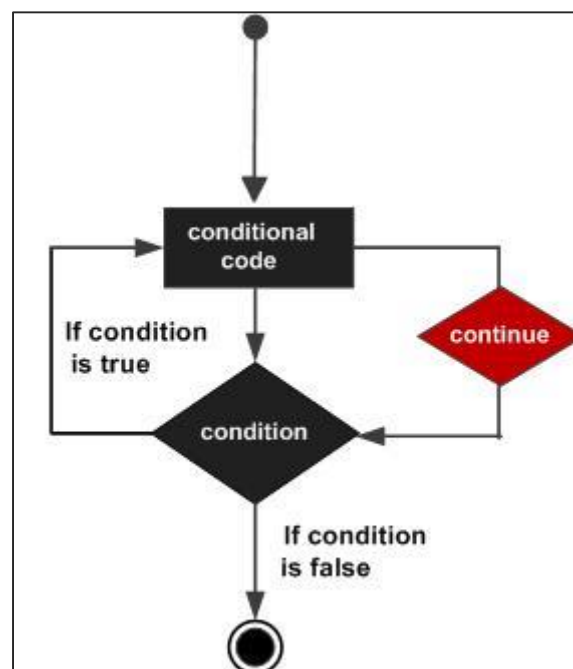
- In a for loop, the continue keyword causes control to immediately jump to the update statement.
- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

Syntax

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Flow Diagram



Example

```
public class Test {  
  
    public static void main(String args[]) {  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ) {  
            if( x == 30 ) {  
                continue;  
            }  
            System.out.print( x );  
            System.out.print("\n");  
        }  
    }  
}
```

This will produce the following result:

```
10  
20  
40  
50
```

Enhanced for loop in Java

As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

Syntax

Following is the syntax of enhanced for loop:

```
for(declaration : expression)  
{  
    //Statements  
}
```

- **Declaration:** The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

- **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example

```
public class Test {  
  
    public static void main(String args[]){  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ){  
            System.out.print( x );  
            System.out.print(",");  
        }  
        System.out.print("\n");  
        String [] names ={"James", "Larry", "Tom", "Lacy"};  
        for( String name : names ) {  
            System.out.print( name );  
            System.out.print(",");  
        }  
    }  
}
```

This will produce the following result:

```
10,20,30,40,50,  
James,Larry,Tom,Lacy,
```

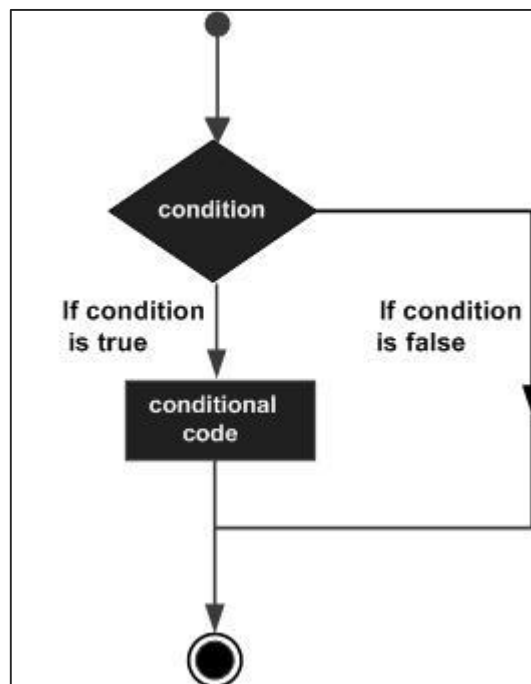
What is Next?

In the following chapter, we will be learning about decision making statements in Java programming.

10. Java – Decision Making

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Java programming language provides following types of decision making statements. Click the following links to check their detail.

Statement	Description
<u>if statement</u>	An if statement consists of a boolean expression followed by one or more statements.
<u>if...else statement</u>	An if statement can be followed by an optional else statement , which executes when the boolean expression is false.

<u>nested if statements</u>	You can use one if or else if statement inside another if or else if statement(s).
<u>switch statement</u>	A switch statement allows a variable to be tested for equality against a list of values.

If Statement in Java

An **if** statement consists of a Boolean expression followed by one or more statements.

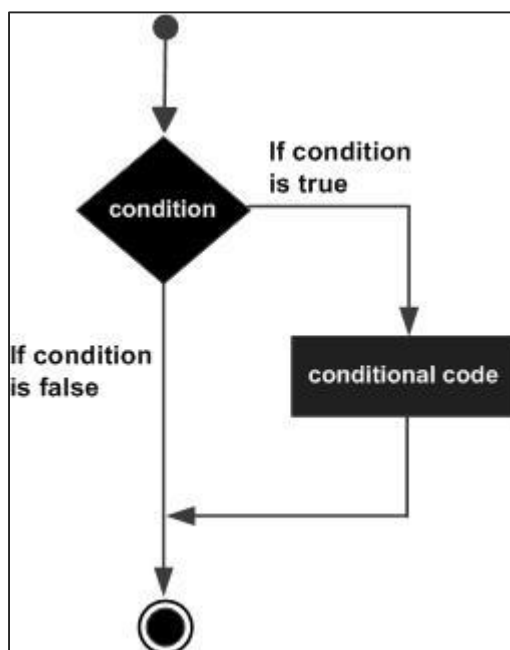
Syntax

Following is the syntax of an if statement:

```
if(Boolean_expression)
{
    //Statements will execute if the Boolean expression is true
}
```

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
public class Test {  
  
    public static void main(String args[]){  
        int x = 10;  
  
        if( x < 20 ){  
            System.out.print("This is if statement");  
        }  
    }  
}
```

This will produce the following result:

```
This is if statement.
```

If-else Statement in Java

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

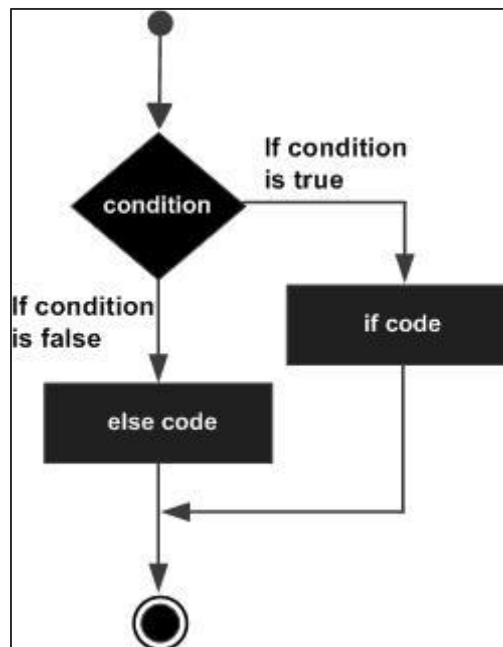
Syntax

Following is the syntax of an if...else statement:

```
if(Boolean_expression){  
    //Executes when the Boolean expression is true  
}else{  
    //Executes when the Boolean expression is false  
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
public class Test {  
  
    public static void main(String args[]){  
        int x = 30;  
  
        if( x < 20 ){  
            System.out.print("This is if statement");  
        }else{  
            System.out.print("This is else statement");  
        }  
    }  
}
```

This will produce the following result:

This is else statement

The if...else if...else Statement

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

Following is the syntax of an if...else statement:

```
if(Boolean_expression 1){
    //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
    //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
    //Executes when the Boolean expression 3 is true
}else {
    //Executes when the none of the above condition is true.
}
```

Example

```
public class Test {

    public static void main(String args[]){
        int x = 30;
        if( x == 10 ){
            System.out.print("Value of X is 10");
        }else if( x == 20 ){
            System.out.print("Value of X is 20");
        }else if( x == 30 ){
            System.out.print("Value of X is 30");
        }else{
            System.out.print("This is else statement");
        }
    }
}
```

```
}
}
```

This will produce the following result:

```
Value of X is 30
```

Nested if Statement in Java

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax

The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1){
    //Executes when the Boolean expression 1 is true
    if(Boolean_expression 2){
        //Executes when the Boolean expression 2 is true
    }
}
```

You can nest **else if...else** in the similar way as we have nested *if* statement.

Example

```
public class Test {

    public static void main(String args[]){
        int x = 30;
        int y = 10;

        if( x == 30 ){
            if( y == 10 ){
                System.out.print("X = 30 and Y = 10");
            }
        }
    }
}
```

This will produce the following result:

X = 30 and Y = 10

Switch Statement in Java

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax of enhanced for loop is:

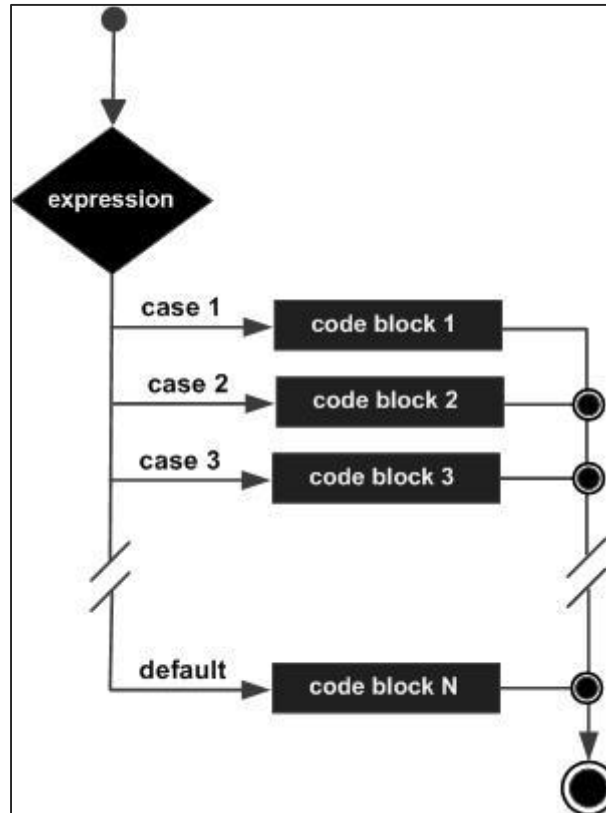
```
switch(expression){
    case value :
        //Statements
        break; //optional
    case value :
        //Statements
        break; //optional
    //You can have any number of case statements.
    default : //Optional
        //Statements
}
```

The following rules apply to a **switch** statement:

- The variable used in a switch statement can only be integers, convertible integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No *break* is needed in the default case.

Flow Diagram



Example

```
public class Test {  
  
    public static void main(String args[]){  
        //char grade = args[0].charAt(0);  
        char grade = 'C';  
  
        switch(grade)  
        {  
            case 'A' :  
                System.out.println("Excellent!");  
                break;  
            case 'B' :  
            case 'C' :  
                System.out.println("Well done");  
        }  
    }  
}
```

```

        break;
    case 'D' :
        System.out.println("You passed");

    case 'F' :
        System.out.println("Better try again");
        break;
    default :
        System.out.println("Invalid grade");
    }
    System.out.println("Your grade is " + grade);
}
}

```

Compile and run the above program using various command line arguments. This will produce the following result:

```

$ java Test
Well done
Your grade is a C
$

```

The ? : Operator:

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

To determine the value of the whole expression, initially exp1 is evaluated.

- If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.
- If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.

What is Next?

In the next chapter, we will discuss about Number class (in the java.lang package) and its subclasses in Java Language.

We will be looking into some of the situations where you will use instantiations of these classes rather than the primitive data types, as well as classes such as formatting, mathematical functions that you need to know about when working with Numbers.

11. Java – Numbers Class

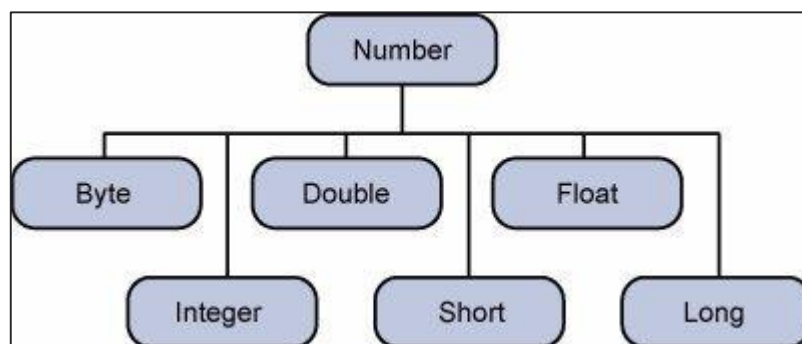
Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

Example

```
int i = 5000;  
float gpa = 13.65;  
byte mask = 0xaf;
```

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 classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.

And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The **Number** class is part of the java.lang package.

Following is an example of boxing and unboxing:

```
public class Test{  
    public static void main(String args[]){  
        Integer x = 5; // boxes int to an Integer object  
        x = x + 10;    // unboxes the Integer to a int  
        System.out.println(x);  
    }  
}
```

This will produce the following result:

15

When x is assigned an integer value, the compiler boxes the integer because x is integer object. Later, x is unboxed so that they can be added as an integer.

Number Methods

Following is the list of the instance methods that all the subclasses of the Number class implements:

Sr. No.	Methods with Description
1	<u>xxxValue()</u> Converts the value of <i>this</i> Number object to the xxx data type and returns it.
2	<u>compareTo()</u> Compares <i>this</i> Number object to the argument.
3	<u>equals()</u> Determines whether <i>this</i> number object is equal to the argument.
4	<u>valueOf()</u> Returns an Integer object holding the value of the specified primitive.
5	<u>toString()</u> Returns a String object representing the value of a specified int or Integer.
6	<u>parseInt()</u> This method is used to get the primitive data type of a certain String.
7	<u>abs()</u> Returns the absolute value of the argument.

8	<u>ceil()</u> Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	<u>floor()</u> Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	<u>rint()</u> Returns the integer that is closest in value to the argument. Returned as a double.
11	<u>round()</u> Returns the closest long or int, as indicated by the method's return type to the argument.
12	<u>min()</u> Returns the smaller of the two arguments.
13	<u>max()</u> Returns the larger of the two arguments.
14	<u>exp()</u> Returns the base of the natural logarithms, e, to the power of the argument.
15	<u>log()</u> Returns the natural logarithm of the argument.
16	<u>pow()</u> Returns the value of the first argument raised to the power of the second argument.

17	<u>sqrt()</u> Returns the square root of the argument.
18	<u>sin()</u> Returns the sine of the specified double value.
19	<u>cos()</u> Returns the cosine of the specified double value.
20	<u>tan()</u> Returns the tangent of the specified double value.
21	<u>asin()</u> Returns the arcsine of the specified double value.
22	<u>acos()</u> Returns the arccosine of the specified double value.
23	<u>atan()</u> Returns the arctangent of the specified double value.
24	<u>atan2()</u> Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.
25	<u>toDegrees()</u> Converts the argument to degrees.
26	<u>toRadians()</u> Converts the argument to radians.

27	<u>random()</u> Returns a random number.
----	--

Java XXXValue Method

Description

The method converts the value of the Number Object that invokes the method to the primitive data type that is returned from the method.

Syntax

Here is a separate method for each primitive data type:

```
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Parameters

Here is the detail of parameters:

- All these are default methods and accepts no parameter.

Return Value

- This method returns the primitive data type that is given in the signature.

Example

```
public class Test{

    public static void main(String args[]){
        Integer x = 5;
        // Returns byte primitive data type
        System.out.println( x.byteValue() );
    }
}
```

```
// Returns double primitive data type
System.out.println(x.doubleValue());

// Returns long primitive data type
System.out.println( x.longValue() );
}
}
```

This will produce the following result:

```
5
5.0
5
```

Java – compareTo() Method

Description

The method compares the Number object that invoked the method to the argument. It is possible to compare Byte, Long, Integer, etc.

However, two different types cannot be compared, both the argument and the Number object invoking the method should be of the same type.

Syntax

```
public int compareTo( NumberSubClass referenceName )
```

Parameters

Here is the detail of parameters:

- **referenceName** -- This could be a Byte, Double, Integer, Float, Long, or Short.

Return Value

- If the Integer is equal to the argument then 0 is returned.
- If the Integer is less than the argument then -1 is returned.
- If the Integer is greater than the argument then 1 is returned.

Example

```
public class Test{

    public static void main(String args[]){
        Integer x = 5;
        System.out.println(x.compareTo(3));
        System.out.println(x.compareTo(5));
        System.out.println(x.compareTo(8));
    }
}
```

This will produce the following result:

```
1
0
-1
```

Java – equals() Method

Description

The method determines whether the Number object that invokes the method is equal to the object that is passed as an argument.

Syntax

```
public boolean equals(Object o)
```

Parameters

Here is the detail of parameters:

- -- Any object.

Return Value

- The method returns True if the argument is not null and is an object of the same type and with the same numeric value. There are some extra requirements for Double and Float objects that are described in the Java API documentation.

Example

```
public class Test{

    public static void main(String args[]){
        Integer x = 5;
        Integer y = 10;
        Integer z =5;
        Short a = 5;

        System.out.println(x.equals(y));
        System.out.println(x.equals(z));
        System.out.println(x.equals(a));
    }
}
```

This will produce the following result:

```
false
true
false
```

Java – valueOf() Method

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

Following are all the variants of this method:

```
static Integer valueOf(int i)
static Integer valueOf(String s)
static Integer 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 the 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);
        System.out.println(c);
        System.out.println(a);
        System.out.println(b);
    }
}
```

This will produce the following result:

```
9
5.0
80.0
1092
```

Java – toString() Method

Description

The method is used to get a String object representing the value of the Number Object.

If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.

If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

Syntax

Following are all the variants of this method:

```
String toString()  
static String toString(int i)
```

Parameters

Here is the detail of parameters:

- **i** -- An int for which string representation would be returned.

Return Value

- **toString():** This returns a String object representing the value of **this**Integer.
- **toString(int i):** This returns a String object representing the specified integer.

Example

```
public class Test{  
  
    public static void main(String args[]){  
        Integer x = 5;  
  
        System.out.println(x.toString());  
        System.out.println(Integer.toString(12));  
    }  
}
```

This will produce the following result:

```
5  
12
```

Java – parseInt() Method

Description

This method is used to get the primitive data type of a certain String. `parseXxx()` is a static method and can have one argument or two.

Syntax

Following are all the variants of this method:

```
static int parseInt(String s)

static int parseInt(String s, int radix)
```

Parameters

Here is the detail of parameters:

- **s** -- This is a string representation of decimal.
- **radix** -- This would be used to convert String **s** into integer.

Return Value

- **parseInt(String s):** This returns an integer (decimal only).
- **parseInt(int i):** This returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.

Example

```
public class Test{

    public static void main(String args[]){
        int x =Integer.parseInt("9");
        double c = Double.parseDouble("5");
        int b = Integer.parseInt("444",16);

        System.out.println(x);
        System.out.println(c);
        System.out.println(b);
    }
}
```

This will produce the following result:

```
9
5.0
1092
```

Java – abs() Method

Description

The method gives the absolute value of the argument. The argument can be int, float, long, double, short, byte.

Syntax

Following are all the variants of this method:

```
double abs(double d)
float abs(float f)
int abs(int i)
long abs(long lng)
```

Parameters

Here is the detail of parameters:

- Any primitive data type

Return Value

- This method Returns the absolute value of the argument.

Example

```
public class Test{

    public static void main(String args[]){
        Integer a = -8;
        double d = -100;
        float f = -90;

        System.out.println(Math.abs(a));
        System.out.println(Math.abs(d));
        System.out.println(Math.abs(f));
    }
}
```



```
}  
}
```

This will produce the following result:

```
8  
100.0  
90.0
```

Java – ceil() Method

Description

The method ceil gives the smallest integer that is greater than or equal to the argument.

Syntax

This method has the following variants:

```
double ceil(double d)  
  
double ceil(float f)
```

Parameters

Here is the detail of parameters:

- A double or float primitive data type

Return Value

- This method returns the smallest integer that is greater than or equal to the argument. Returned as a double.

Example

```
public class Test{  
  
    public static void main(String args[]){  
        double d = -100.675;  
        float f = -90;  
  
        System.out.println(Math.ceil(d));  
        System.out.println(Math.ceil(f));  
    }  
}
```

```
        System.out.println(Math.floor(d));  
        System.out.println(Math.floor(f));  
    }  
}
```

This will produce the following result:

```
-100.0  
-90.0  
-101.0  
-90.0
```

Java – floor() Method

Description

The method floor gives the largest integer that is less than or equal to the argument.

Syntax

This method has the following variants:

```
double floor(double d)  
  
double floor(float f)
```

Parameters

Here is the detail of parameters:

- A double or float primitive data type.

Return Value

- This method returns the largest integer that is less than or equal to the argument. Returned as a double.

Example

```
public class Test{  
  
    public static void main(String args[]){  
        double d = -100.675;  
        float f = -90;
```

```
        System.out.println(Math.floor(d));  
        System.out.println(Math.floor(f));  
  
        System.out.println(Math.ceil(d));  
        System.out.println(Math.ceil(f));  
    }  
}
```

This will produce the following result:

```
-101.0  
-90.0  
-100.0  
-90.0
```

Java – rint() Method

Description

The method rint returns the integer that is closest in value to the argument.

Syntax

```
double rint(double d)
```

Parameters

Here is the detail of parameters:

- **d** -- it accepts a double value as parameter.

Return Value

- This method returns the integer that is closest in value to the argument. Returned as a double.

Example

```
public class Test{  
    public static void main(String args[]){  
        double d = 100.675;  
        double e = 100.500;  
        double f = 100.200;
```

```
        System.out.println(Math rint(d));  
        System.out.println(Math rint(e));  
        System.out.println(Math rint(f));  
    }  
}
```

This will produce the following result:

```
101.0  
100.0  
100.0
```

Java – round() Method

Description

The method round returns the closest long or int, as given by the methods return type.

Syntax

This method has the following variants:

```
long round(double d)  
  
int round(float f)
```

Parameters

Here is the detail of parameters:

- **d** -- A double or float primitive data type
- **f** -- A float primitive data type

Return Value

- This method returns the closest long or int, as indicated by the method's return type, to the argument.

Example

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

```
double d = 100.675;
double e = 100.500;
float f = 100;
float g = 90f;

System.out.println(Math.round(d));
System.out.println(Math.round(e));
System.out.println(Math.round(f));
System.out.println(Math.round(g));
}
}
```

This will produce the following result:

```
101
101
100
90
```

Java – min() Method

Description

The method gives the smaller of the two arguments. The argument can be int, float, long, double.

Syntax

This method has the following variants:

```
double min(double arg1, double arg2)
float min(float arg1, float arg2)
int min(int arg1, int arg2)
long min(long arg1, long arg2)
```

Parameters

Here is the detail of parameters:

- This method accepts any primitive data type as a parameter.

Return Value

- This method returns the smaller of the two arguments.

Example

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

This will produce the following result:

```
12.123  
23.0
```

Java – max() Method

Description

This method gives the maximum of the two arguments. The argument can be int, float, long, double.

Syntax

This method has the following variants:

```
double max(double arg1, double arg2)  
float max(float arg1, float arg2)  
int max(int arg1, int arg2)  
long max(long arg1, long arg2)
```

Parameters

Here is the detail of parameters:

- This method accepts any primitive data type as a parameter.

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 will produce the following result:

```
12.456
23.12
```

Java – exp() Method

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** --Any primitive data type.

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 will produce the following result:

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

Java – log() Method

Description

The method returns the natural logarithm of the argument.

Syntax

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

Parameters

Here is the detail of parameters:

- **d** -- Any primitive data type.

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 will produce the following result:

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

Java – pow() Method

Description

The method returns the value of the first argument raised to the power of the second argument.

Syntax

```
double pow(double base, double exponent)
```

Parameters

Here is the detail of parameters –

- **base** -- Any primitive data type.
- **exponent** -- Any primitive data type.

Return Value

- This method returns the value of the first argument raised to the power of the second 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("pow(%.3f, %.3f) is %.3f%n", x, y, Math.pow(x, y));

    }
}
```

This will produce the following result –

```
The value of e is 2.7183
pow(11.635, 2.760) is 874.008
```

Java – sqrt() Method

Description

The method returns the square root of the argument.

Syntax

```
double sqrt(double d)
```

Parameters

Here is the detail of parameters:

- **d** -- Any primitive data type.

Return Value

- This method returns the square root 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("sqrt(%.3f) is %.3f%n", x, Math.sqrt(x));
    }
}
```

This will produce the following result:

```
The value of e is 2.7183
sqrt(11.635) is 3.411
```

Java – sin() Method

Description

The method returns the sine of the specified double value.

Syntax

```
double sin(double d)
```

Parameters

Here is the detail of parameters:

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

Return Value

- This method returns the sine 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 sine of %.1f degrees is %.4f%n", degrees,
Math.sin(radians));

    }
}
```

This will produce the following result:

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

Java – cos() Method

Description

The method returns the cosine of the specified double value.

Syntax

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

Parameters

Here is the detail of parameters:

- **d** -- This method accepts a value of double data type.

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 will produce the following result:

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

Java – tan() Method

Description

The method returns the tangent of the specified double value.

Syntax

```
double tan(double d)
```

Parameters

Here is the detail of parameters:

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

Return Value

- This method returns the tangent 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 tangent of %.1f degrees is %.4f%n", degrees,
Math.tan(radians));

    }
}
```

This will produce the following result:

```
The value of pi is 3.1416
The tangent of 45.0 degrees is 1.0000
```

Java – asin() Method

Description

The method returns the arcsine of the specified double value.

Syntax

```
double asin(double d)
```

Parameters

Here is the detail of parameters:

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

Return Value

- This method returns the arcsine 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 arcsine of %.4f is %.4f degrees %n",
Math.sin(radians), Math.toDegrees(Math.asin(Math.sin(radians))));

    }
}
```

This will produce the following result:

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

Java – acos() Method

Description

The method returns the arccosine of the specified double value.

Syntax

```
double acos(double d)
```

Parameters

Here is the detail of parameters:

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

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 will produce the following result:

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

Java – atan() Method

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 type.

Return Value

- This method returns the arctangent 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 arctangent of %.4f is %.4f degrees %n",
Math.cos(radians), Math.toDegrees(Math.atan(Math.sin(radians))));

    }
}
```

This will produce the following result:

```
The value of pi is 3.1416
The arctangent of 1.0000 is 45.0000 degrees
```

Java – atan2() Method

Description

The method converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.

Syntax

```
double atan2(double y, double x)
```

Parameters

Here is the detail of parameters:

- **X** -- X co-ordinate in double data type.
- **Y** -- Y co-ordinate in double data type.

Return Value

- This method returns theta from polar coordinate (r, theta).

Example

```
public class Test{

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

        System.out.println( Math.atan2(x, y) );
    }
}
```

This will produce the following result:

```
0.982793723247329
```

Java – toDegrees() Method

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 will produce the following result:

```
2578.3100780887044
1718.8733853924698
```

Java – toRadians() Method

Description

The method converts the argument value to radians.

Syntax

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

This will produce the following result:

```
0.7853981633974483
0.5235987755982988
```

Java – random() Method

Description

The method is used to generate a random number between 0.0 and 1.0. The range is: $0.0 \leq \text{Math.random} < 1.0$. Different ranges can be achieved by using arithmetic operations.

Syntax

```
static double random()
```

Parameters

Here is the detail of parameters:

- This is a default method and accepts no parameter.

Return Value

- This method returns a double.

Example

```
public class Test{  
  
    public static void main(String args[]){  
        System.out.println( Math.random() );  
        System.out.println( Math.random() );  
    }  
}
```

This will produce the following result:

```
0.16763945061451657  
0.400551253762343
```

Note: The above result will vary every time you call random() method.

What is Next?

In the next section, we will be going through the Character class in Java. You will be learning how to use object Characters and primitive data type char in Java.

12. Java – Character Class

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 = new Character('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 a special meaning to the compiler.

The newline character (`\n`) has been used frequently in this tutorial in `System.out.println()` statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences:

Escape Sequence	Description
<code>\t</code>	Inserts a tab in the text at this point.
<code>\b</code>	Inserts a backspace in the text at this point.
<code>\n</code>	Inserts a newline in the text at this point.
<code>\r</code>	Inserts a carriage return in the text at this point.
<code>\f</code>	Inserts a form feed in the text at this point.
<code>\'</code>	Inserts a single quote character in the text at this point.
<code>\"</code>	Inserts a double quote character in the text at this point.
<code>\\</code>	Inserts a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Example

If you want to put quotes within quotes, you must use the escape sequence, `\"`, on the interior quotes:

```
public class Test {

    public static void main(String args[]) {
        System.out.println("She said \"Hello!\" to me.");
    }
}
```

This will produce the following result:

```
She said "Hello!" to me.
```

Character Methods

Following is the list of the important instance methods that all the subclasses of the Character class implement:

Sr. No.	Methods with Description
1	<u>isLetter()</u> Determines whether the specified char value is a letter.
2	<u>isDigit()</u> Determines whether the specified char value is a digit.
3	<u>isWhitespace()</u> Determines whether the specified char value is white space.
4	<u>isUpperCase()</u> Determines whether the specified char value is uppercase.
5	<u>isLowerCase()</u> Determines whether the specified char value is lowercase.
6	<u>toUpperCase()</u> Returns the uppercase form of the specified char value.
7	<u>toLowerCase()</u> Returns the lowercase form of the specified char value.
8	<u>toString()</u> Returns a String object representing the specified character value that is, a one-character string.

Java – isLetter() Method

Description

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

Syntax

```
boolean isLetter(char ch)
```


Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns true if the passed character is really a character.

Example

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

This will produce the following result:

```
true  
false
```

Java – isDigit() Method

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 the 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 will produce the following result:

```
false  
true
```

Java – isWhitespace() Method

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)
```

Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns true, if the passed character is really a white space.

Example

```
public class Test{

    public static void main(String args[]){
        System.out.println(Character.isWhitespace('c'));
        System.out.println(Character.isWhitespace(' '));

        System.out.println(Character.isWhitespace('\n'));
        System.out.println(Character.isWhitespace('\t'));
    }
}
```

This will produce the following result:

```
false
true
true
true
```

Java – isUpperCase() Method

Description

This method determines whether the specified char value is uppercase.

Syntax

```
boolean isUpperCase(char ch)
```

Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns true, if the passed character is really an uppercase.

Example

```
public class Test{  
  
    public static void main(String args[]){  
        System.out.println( Character.isUpperCase('c'));  
        System.out.println( Character.isUpperCase('C'));  
        System.out.println( Character.isUpperCase('\n'));  
        System.out.println( Character.isUpperCase('\t'));  
    }  
}
```

This will produce the following result:

```
false  
true  
false  
false
```

Java – isLowerCase() Method

Description

The method determines whether the specified char value is lowercase.

Syntax

```
boolean isLowerCase(char ch)
```

Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns true, if the passed character is really in lowercase.

Example

```
public class Test{

    public static void main(String args[]){
        System.out.println(Character.isLowerCase('c'));
        System.out.println(Character.isLowerCase('C'));
        System.out.println(Character.isLowerCase('\n'));
        System.out.println(Character.isLowerCase('\t'));
    }
}
```

This will produce the following result:

```
true
false
false
false
```

Java – toUpperCase() Method

Description

The method returns the uppercase form of the specified char value.

Syntax

```
char toUpperCase(char ch)
```

Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns the uppercase form of the specified char value.

Example

```
public class Test{

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

This will produce the following result:

```
C
C
```

Java – toLowerCase() Method

Description

The method returns the lowercase form of the specified char value.

Syntax

```
char toLowerCase(char ch)
```

Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns the lowercase form of the specified char value.

Example

```
public class Test{

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

This will produce the following result:

```
c
c
```

Java – toString() Method

Description

This method returns a String object representing the specified character value, that is, a one-character string.

Syntax

```
String toString(char ch)
```

Parameters

Here is the detail of parameters:

- **ch** -- Primitive character type.

Return Value

- This method returns String object.

Example

```
public class Test{  
  
    public static void main(String args[]){  
        System.out.println(Character.toString('c'));  
        System.out.println(Character.toString('C'));  
    }  
}
```

This will produce the following result:

```
c  
C
```

For a complete list of methods, please refer to the [java.lang.Character API specification](#).

What is Next?

In the next section, we will be going through the String class in Java. You will be learning how to declare and use Strings efficiently as well as some of the important methods in the String class.

13. Java – Strings Class

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!".

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

```
public class StringDemo{

    public static void main(String args[]){
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
        String helloString = new String(helloArray);
        System.out.println( helloString );
    }
}
```

This will produce the following result:

```
hello.
```

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](#).

Java – String Buffer & String Builder Classes

The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.

Unlike Strings, objects of type StringBuffer and String builder can be modified over and over again without leaving behind a lot of new unused objects.

The `StringBuilder` class was introduced as of Java 5 and the main difference between the `StringBuffer` and `StringBuilder` is that `StringBuilders` methods are not thread safe (not synchronised).

It is recommended to use **`StringBuilder`** whenever possible because it is faster than `StringBuffer`. However, if the thread safety is necessary, the best option is `StringBuffer` objects.

Example

```
public class Test{

    public static void main(String args[]){
        StringBuffer sBuffer = new StringBuffer(" test");
        sBuffer.append(" String Buffer");
        System.out.println(sBuffer);
    }
}
```

This will produce the following result:

```
test String Buffer
```

StringBuffer Methods

Here is the list of important methods supported by `StringBuffer` class:

Sr. No.	Methods with Description
1	<u><code>public StringBuffer append(String s)</code></u> Updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings, etc.
2	<u><code>public StringBuffer reverse()</code></u> The method reverses the value of the <code>StringBuffer</code> object that invoked the method.
3	<u><code>public delete(int start, int end)</code></u> Deletes the string starting from the start index until the end index.

4	<u>public insert(int offset, int i)</u> This method inserts a string s at the position mentioned by the offset.
5	<u>replace(int start, int end, String str)</u> This method replaces the characters in a substring of this StringBuffer with characters in the specified String.

Java – String Buffer append() Method

Description

This method updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings, etc.

Syntax

Here is a separate method for each primitive data type:

```
public StringBuffer append(boolean b)
public StringBuffer append(char c)
public StringBuffer append(char[] str)
public StringBuffer append(char[] str, int offset, int len)
public StringBuffer append(double d)
public StringBuffer append(float f)
public StringBuffer append(int i)
public StringBuffer append(long l)
public StringBuffer append(Object obj)
public StringBuffer append(StringBuffer sb)
public StringBuffer append(String str)
```

Parameters

Here is the detail of parameters:

- Here the parameter depends on what you are trying to append in the String Buffer.

Return Value

- These methods return the updated StringBuffer objects.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("Test");  
        sb.append(" String Buffer");  
        System.out.println(sb);  
    }  
}
```

This will produce the following result:

```
Test String Buffer
```

Java – String Buffer reverse() Method

Description

This method reverses the value of the StringBuffer object that invoked the method.

Let n be the length of the old character sequence, the one contained in the string buffer just prior to the execution of the reverse method. Then, the character at index k in the new character sequence is equal to the character at index $n-k-1$ in the old character sequence.

Syntax

Here is the syntax for this method:

```
public StringBuffer reverse()
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- This method returns StringBuffer object with the reversed sequence.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        StringBuffer buffer = new StringBuffer("Game Plan");  
        buffer.reverse();  
        System.out.println(buffer);  
    }  
}
```

This will produce the following result:

```
nalP emaG
```

Java – String Buffer delete() Method

Description

This method removes the characters in a substring of this StringBuffer. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the StringBuffer if no such character exists.

If start is equal to end, no changes are made.

Syntax

Here is the syntax of this method:

```
public StringBuffer delete(int start, int end)
```

Parameters

Here is the detail of parameters:

- **start** -- The beginning index, inclusive.
- **end** -- The ending index, exclusive.

Return Value

- This method returns the StringBuffer object.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("abcdefghijk");  
        sb.delete(3,7);  
        System.out.println(sb);  
    }  
}
```

This will produce the following result:

```
abchijk
```

Java – String Buffer insert() Method

Description

This method removes the characters in a substring of this StringBuffer. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the StringBuffer, if no such character exists.

If start is equal to end, no changes are made.

Syntax

Here is a separate method for each primitive data type:

```
public StringBuffer insert(int offset, boolean b)  
public StringBuffer insert(int offset, char c)  
public insert(int offset, char[] str)  
public StringBuffer insert(int index, char[] str,  
                           int offset, int len)  
public StringBuffer insert(int offset, float f)  
public StringBuffer insert(int offset, int i)  
public StringBuffer insert(int offset, long l)  
public StringBuffer insert(int offset, Object obj)  
public StringBuffer insert(int offset, String str)
```

Parameters

Here is the detail of parameters:

- Parameter depends on what you are trying to insert.

Return Value

- This method returns the modified StringBuffer object.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("abcdefghijk");  
        sb.insert(3,"123");  
        System.out.println(sb);  
    }  
}
```

This will produce the following result:

```
abc123defghijk
```

Java – String Buffer replace() Method

Description

This method replaces the characters in a substring of this StringBuffer with characters in the specified String.

The substring begins at the specified start and extends to the character at index end - 1 or to the end of the StringBuffer, if no such character exists. First the characters in the substring are removed and then the specified String is inserted at start.

Syntax

Here is the syntax of this method:

```
public StringBuffer replace(int start, int end, String str)
```

Parameters

Here is the detail of parameters:

- **start** -- The beginning index, inclusive.
- **end** -- The ending index, exclusive.
- **str** -- String that will replace previous contents.

Return Value

- This method returns the modified StringBuffer object.

Example

```
public class Test {

    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("abcdefghijk");
        sb.replace(3, 8, "ZARA");
        System.out.println(sb);
    }
}
```

This will produce the following result:

```
abcZARAIjk
```

Here is the list of other methods (except set methods) which are very similar to String class:

Sr. No.	Methods with Description
1	int capacity() Returns the current capacity of the String buffer.

2	char charAt(int index) The specified character of the sequence currently represented by the string buffer, as indicated by the index argument, is returned.
3	void ensureCapacity(int minimumCapacity) Ensures that the capacity of the buffer is at least equal to the specified minimum.
4	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Characters are copied from this string buffer into the destination character array dst.
5	int indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
6	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.
7	int lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
8	int lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring.
9	int length() Returns the length (character count) of this string buffer.
10	void setCharAt(int index, char ch) The character at the specified index of this string buffer is set to ch.

11	void setLength(int newLength) Sets the length of this String buffer.
12	CharSequence subSequence(int start, int end) Returns a new character sequence that is a subsequence of this sequence.
13	String substring(int start) Returns a new String that contains a subsequence of characters currently contained in this StringBuffer. The substring begins at the specified index and extends to the end of the StringBuffer.
14	String substring(int start, int end) Returns a new String that contains a subsequence of characters currently contained in this StringBuffer.
15	String toString() Converts to a string representing the data in this string buffer.

String Length

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

The following program is an example of **length()**, method String class.

```
public class StringDemo {

    public static void main(String args[]) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        System.out.println( "String Length is : " + len );
    }
}
```

This will produce the following result:

```
String Length is : 17
```

Concatenating Strings

The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

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

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in:

```
"Hello," + " world" + "!"
```

which results in:

```
"Hello, world!"
```

Let us look at the following example:

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String string1 = "saw I was ";  
        System.out.println("Dot " + string1 + "Tod");  
    }  
}
```

This will produce the following result:

```
Dot saw I was Tod
```

Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of:

```
System.out.printf("The value of the float variable is " +
    "%f, while the value of the integer " +
    "variable is %d, and the string " +
    "is %s", floatVar, intVar, stringVar);
```

You can write:

```
String fs;
fs = String.format("The value of the float variable is " +
    "%f, while the value of the integer " +
    "variable is %d, and the string " +
    "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String Methods

Here is the list of methods supported by String class:

Sr. No.	Methods with Description
1	<u>char charAt(int index)</u> Returns the character at the specified index.
2	<u>int compareTo(Object o)</u> Compares this String to another Object.
3	<u>int compareTo(String anotherString)</u> Compares two strings lexicographically.
4	<u>int compareToIgnoreCase(String str)</u> Compares two strings lexicographically, ignoring case differences.

5	<u>String concat(String str)</u> Concatenates the specified string to the end of this string.
6	<u>boolean contentEquals(StringBuffer sb)</u> Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	<u>static String copyValueOf(char[] data)</u> Returns a String that represents the character sequence in the array specified.
8	<u>static String copyValueOf(char[] data, int offset, int count)</u> Returns a String that represents the character sequence in the array specified.
9	<u>boolean endsWith(String suffix)</u> Tests if this string ends with the specified suffix.
10	<u>boolean equals(Object anObject)</u> Compares this string to the specified object.
11	<u>boolean equalsIgnoreCase(String anotherString)</u> Compares this String to another String, ignoring case considerations.
12	<u>byte getBytes()</u> Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	<u>byte[] getBytes(String charsetName)</u> Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	<u>void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)</u> Copies characters from this string into the destination character array.

15	<u>int hashCode()</u> Returns a hash code for this string.
16	<u>int indexOf(int ch)</u> Returns the index within this string of the first occurrence of the specified character.
17	<u>int indexOf(int ch, int fromIndex)</u> Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	<u>int indexOf(String str)</u> Returns the index within this string of the first occurrence of the specified substring.
19	<u>int indexOf(String str, int fromIndex)</u> Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	<u>String intern()</u> Returns a canonical representation for the string object.
21	<u>int lastIndexOf(int ch)</u> Returns the index within this string of the last occurrence of the specified character.
22	<u>int lastIndexOf(int ch, int fromIndex)</u> Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	<u>int lastIndexOf(String str)</u> Returns the index within this string of the rightmost occurrence of the specified substring.
24	<u>int lastIndexOf(String str, int fromIndex)</u> Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

25	<u>int length()</u> Returns the length of this string.
26	<u>boolean matches(String regex)</u> Tells whether or not this string matches the given regular expression.
27	<u>boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)</u> Tests if two string regions are equal.
28	<u>boolean regionMatches(int toffset, String other, int ooffset, int len)</u> Tests if two string regions are equal.
29	<u>String replace(char oldChar, char newChar)</u> Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	<u>String replaceAll(String regex, String replacement)</u> Replaces each substring of this string that matches the given regular expression with the given replacement.
31	<u>String replaceFirst(String regex, String replacement)</u> Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	<u>String[] split(String regex)</u> Splits this string around matches of the given regular expression.
33	<u>String[] split(String regex, int limit)</u> Splits this string around matches of the given regular expression.
34	<u>boolean startsWith(String prefix)</u> Tests if this string starts with the specified prefix.

35	<u>boolean startsWith(String prefix, int toffset)</u> Tests if this string starts with the specified prefix beginning a specified index.
36	<u>CharSequence subSequence(int beginIndex, int endIndex)</u> Returns a new character sequence that is a subsequence of this sequence.
37	<u>String substring(int beginIndex)</u> Returns a new string that is a substring of this string.
38	<u>String substring(int beginIndex, int endIndex)</u> Returns a new string that is a substring of this string.
39	<u>char[] toCharArray()</u> Converts this string to a new character array.
40	<u>String toLowerCase()</u> Converts all of the characters in this String to lower case using the rules of the default locale.
41	<u>String toLowerCase(Locale locale)</u> Converts all of the characters in this String to lower case using the rules of the given Locale.
42	<u>String toString()</u> This object (which is already a string!) is itself returned.
43	<u>String toUpperCase()</u> Converts all of the characters in this String to upper case using the rules of the default locale.
44	<u>String toUpperCase(Locale locale)</u> Converts all of the characters in this String to upper case using the rules of the given Locale.

45	<u>String trim()</u> Returns a copy of the string, with leading and trailing whitespace omitted.
46	<u>static String valueOf(primitive data type x)</u> Returns the string representation of the passed data type argument.

Java – String charAt() Method

Description

This method returns the character located at the String's specified index. The string indexes start from zero.

Syntax

Here is the syntax of this method:

```
public char charAt(int index)
```

Parameters

Here is the detail of parameters:

- **index** -- Index of the character to be returned.

Return Value

- This method returns a char at the specified index.

Example

```
public class Test {

    public static void main(String args[]) {
        String s = "Strings are immutable";
        char result = s.charAt(8);
        System.out.println(result);
    }
}
```

This will produce the following result:

```
a
```

Java – String compareTo(Object o) Method

Description

This method compares this String to another Object.

Syntax

Here is the syntax of this method:

```
int compareTo(Object o)
```

Parameters

Here is the detail of parameters:

- **O**-- the Object to be compared.

Return Value

- The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        String str1 = "Strings are immutable";  
        String str2 = new String("Strings are immutable");  
        String str3 = new String("Integers are not immutable");  
  
        int result = str1.compareTo( str2 );  
        System.out.println(result);  
  
        result = str2.compareTo( str3 );  
        System.out.println(result);  
  
    }  
}
```

This will produce the following result:

```
0
10
```

Java – String compareTo(String anotherString) Method

Description

This method compares two strings lexicographically.

Syntax

Here is the syntax of this method:

```
int compareTo(String anotherString)
```

Parameters

Here is the detail of parameters:

- **anotherString** -- the String to be compared.

Return Value

- The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

Example

```
public class Test {

    public static void main(String args[]) {
        String str1 = "Strings are immutable";
        String str2 = "Strings are immutable";
        String str3 = "Integers are not immutable";

        int result = str1.compareTo( str2 );
        System.out.println(result);

        result = str2.compareTo( str3 );
        System.out.println(result);
    }
}
```

```
        result = str3.compareTo( str1 );
        System.out.println(result);
    }
}
```

This will produce the following result:

```
0
10
-10
```

Java – String compareToIgnoreCase() Method

Description

This method compares two strings lexicographically, ignoring case differences.

Syntax

Here is the syntax of this method:

```
int compareToIgnoreCase(String str)
```

Parameters

Here is the detail of parameters:

- **str** -- the String to be compared.

Return Value

- This method returns a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations.

Example

```
public class Test {

    public static void main(String args[]) {
        String str1 = "Strings are immutable";
        String str2 = "Strings are immutable";
        String str3 = "Integers are not immutable";
```

```
int result = str1.compareToIgnoreCase( str2 );
System.out.println(result);

result = str2.compareToIgnoreCase( str3 );
System.out.println(result);

result = str3.compareToIgnoreCase( str1 );
System.out.println(result);
}
}
```

This will produce the following result:

```
0
10
-10
```

Java – String concat() Method

Description

This method appends one String to the end of another. The method returns a String with the value of the String passed into the method, appended to the end of the String, used to invoke this method.

Syntax

Here is the syntax of this method:

```
public String concat(String s)
```

Parameters

Here is the detail of parameters:

- **s** -- the String that is concatenated to the end of this String.

Return Value

- This methods returns a string that represents the concatenation of this object's characters followed by the string argument's characters.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        String s = "Strings are immutable";  
        s = s.concat(" all the time");  
        System.out.println(s);  
    }  
}
```

This will produce the following result:

```
Strings are immutable all the time
```

Java – String contentEquals() Method

Description

This method returns true if and only if this String represents the same sequence of characters as specified in StringBuffer.

Syntax

Here is the syntax of this method:

```
public boolean contentEquals(StringBuffer sb)
```

Parameters

Here is the detail of parameters:

- **sb** -- the StringBuffer to compare.

Return Value

- This method returns true if and only if this String represents the same sequence of characters as the specified in StringBuffer, otherwise false.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        String str1 = "Not immutable";  
        String str2 = "Strings are immutable";  
        StringBuffer str3 = new StringBuffer( "Not immutable");  
  
        boolean result = str1.contentEquals( str3 );  
        System.out.println(result);  
  
        result = str2.contentEquals( str3 );  
        System.out.println(result);  
    }  
}
```

This will produce the following result:

```
true  
false
```

Java – String copyValueOf(char[] data) Method

Description

This method returns a String that represents the character sequence in the array specified.

Syntax

Here is the syntax of this method:

```
public static String copyValueOf(char[] data)
```

Parameters

Here is the detail of parameters:

- **data** -- the character array.

Return Value

- This method returns a String that contains the characters of the character array.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};  
        String Str2 = "";  
  
        Str2 = Str2.copyValueOf( Str1 );  
        System.out.println("Returned String: " + Str2);  
  
    }  
}
```

This will produce the following result:

```
Returned String: hello world
```

Java – String copyValueOf(char[] data, int offset, int count) Method

Description

This returns a String that represents the character sequence in the array specified.

Syntax

Here is the syntax of this method:

```
public static String copyValueOf(char[] data, int offset, int count)
```

Parameters

Here is the detail of parameters:

- **data** -- the character array.
- **offset** -- initial offset of the subarray.
- **count** -- length of the subarray.

Return Value

- This method returns a String that contains the characters of the character array.

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};  
        String Str2 = "";  
  
        Str2 = Str2.copyValueOf( Str1, 2, 6 );  
        System.out.println("Returned String: " + Str2);  
    }  
}
```

This will produce the following result:

```
Returned String: llo wo
```

Java – String endsWith() Method

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 will produce the following result:

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

Java – String equals() Method

Description

This method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Syntax

Here is the syntax of this method:

```
public boolean equals(Object anObject)
```

Parameters

Here is the detail of parameters:

- **anObject** -- the object to compare this String against.

Return Value

- This method returns true if the String are equal; 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!!");  
        boolean retVal;  
  
        retVal = Str1.equals( Str2 );  
        System.out.println("Returned Value = " + retVal );  
  
        retVal = Str1.equals( Str3 );  
        System.out.println("Returned Value = " + retVal );  
    }  
}
```

This will produce the following result:

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

Java – String equalsIgnoreCase() Method

Description

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 will produce the following result:

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

Java – String getBytes(String charsetName) Method

This method encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

Syntax

Here is the syntax of this method:

```
public byte[] getBytes(String charsetName) throws UnsupportedOperationException
```

Parameters

Here is the detail of parameters:

- **charsetName** -- the name of a supported charset.

Return Value

- This method returns the resultant byte array.

Example

```
import java.io.*;

public class Test{

    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");

        try{
            Str2 = Str1.getBytes( "UTF-8" );
            System.out.println("Returned Value " + Str2 );

            Str2 = Str1.getBytes( "ISO-8859-1" );
            System.out.println("Returned Value " + Str2 );
        }catch( UnsupportedOperationException e){
            System.out.println("Unsupported character set");
        }
    }
}
```

This will produce the following result:

```
Returned Value [B@15ff48b
Returned Value [B@1b90b39
```

Java – String getBytes() Method

Description

This method encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

Syntax

Here is the syntax of this method:

```
public byte[] getBytes()
```

Return Value

- This method returns the resultant byte array.

Example

```
import java.io.*;

public class Test{

    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");

        try{
            byte[] Str2 = Str1.getBytes();
            System.out.println("Returned Value " + Str2 );
        }catch( UnsupportedOperationException e){
            System.out.println("Unsupported character set");
        }
    }
}
```

This will produce the following result:

```
Returned Value [B@192d342
```

Java – String getChars() Method

Description

This method copies characters from this string into the destination character array.

Syntax

Here is the syntax of this method:

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Parameters

Here is the detail of parameters:

- **srcBegin** -- index of the first character in the string to copy.
- **srcEnd** -- index after the last character in the string to copy.
- **dst** -- the destination array.
- **dstBegin** -- the start offset in the destination array.

Return Value

- It does not return any value but throws `IndexOutOfBoundsException`.

Example

```
import java.io.*;

public class Test{
    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");
        char[] Str2 = new char[7];

        try{
            Str1.getChars(2, 9, Str2, 0);
            System.out.print("Copied Value = " );
            System.out.println(Str2 );
        }
    }
}
```

```

        }catch( Exception ex){
            System.out.println("Raised exception...");
        }
    }
}

```

This will produce the following result:

```
Copied Value = lcome t
```

Java – String hashCode() Method

Description

This method returns a hash code for this string. The hash code for a String object is computed as:

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

Using int arithmetic, where $s[i]$ is the i th character of the string, n is the length of the string, and $^$ indicates exponentiation. (The hash value of the empty string is zero.)

Syntax

Here is the syntax of this method:

```
public int hashCode()
```

Parameters

Here is the detail of parameters:

- This is a default method and this will not accept any parameters.

Return Value

- This method returns a hash code value for this object.

Example

```
import java.io.*;

public class Test{
    public static void main(String args[]){
        String Str = new String("Welcome to Tutorialspoint.com");
        System.out.println("Hashcode for Str :" + Str.hashCode() );
    }
}
```

This will produce the following result:

```
Hashcode for Str :1186874997
```

Java – String indexOf(int ch) Method

Description

This method returns the index within this string of the first occurrence of the specified character or -1, if the character does not occur.

Syntax

Here is the syntax of this method:

```
public int indexOf(int ch )
```

Parameters

Here is the detail of parameters:

- **ch** -- a character.

Return Value

- See the description.

Example

```
import java.io.*;
public class Test {

    public static void main(String args[]) {
        String Str = new String("Welcome to Tutorialspoint.com");
        System.out.print("Found Index : " );
        System.out.println(Str.indexOf( 'o' ));
    }
}
```

This will produce the following result:

```
Found Index :4
```

Java – String indexOf(int ch, int fromIndex) Method

Description

This method 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.

Syntax

Here is the syntax of this method:

```
public int indexOf(int ch, int fromIndex)
```

Parameters

Here is the detail of parameters:

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

Return Value

- See the description.

Example

```
import java.io.*;

public class Test {

    public static void main(String args[]) {

        String Str = new String("Welcome to Tutorialspoint.com");
        System.out.print("Found Index : " );
        System.out.println(Str.indexOf( 'o', 5 ));
    }
}
```

This will produce the following result:

```
Found Index :9
```

Java – String indexOf(String str) Method

Description

This method 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.

Syntax

Here is the syntax of this method:

```
int indexOf(String str)
```

Parameters

Here is the detail of parameters:

- **str** -- a string.

Return Value

- See the description.

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");
        System.out.println( Str.indexOf( SubStr1 ));

    }
}
```

This will produce the following result:

```
Found Index :11
```

Java – String indexOf(String str, int fromIndex) Method

This method 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:

```
int indexOf(String str, int fromIndex)
```

Parameters

Here is the detail of parameters:

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

Return Value

- See the description.

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" );
        System.out.print("Found Index : " );
        System.out.println( Str.indexOf( SubStr1, 15 ));

    }

}
```

This will produce the following result:

```
Found Index :-1
```

Java – String Intern() Method

Description

This method returns a canonical representation for the string object. It follows that for any two strings **s** and **t**, **s.intern() == t.intern()** is true if and only if **s.equals(t)** is true.

Syntax

Here is the syntax of this method:

```
public String intern()
```

Parameters

Here is the detail of parameters:

- This is a default method and this do not accept any parameters.

Return Value

- This method returns a canonical representation for the string object.

Example

```
import java.io.*;

public class Test{
    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");
        String Str2 = new String("WELCOME TO SUTORIALSPOINT.COM");

        System.out.print("Canonical representation:" );
        System.out.println(Str1.intern());

        System.out.print("Canonical representation:" );
        System.out.println(Str2.intern());
    }
}
```

This will produce the following result:

```
Canonical representation: Welcome to Tutorialspoint.com
Canonical representation: WELCOME TO SUTORIALSPOINT.COM
```

Java – String lastIndexOf(int ch) Method

Description

This method returns the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.

Syntax

Here is the syntax of this method:

```
int lastIndexOf(int ch)
```

Parameters

Here is the detail of parameters:

- **ch** -- a character.

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");
        System.out.print("Found Last Index : " );
        System.out.println(Str.lastIndexOf( 'o' ));
    }
}
```

This will produce the following result:

```
Found Last Index :27
```

Java – String lastIndexOf(int ch, int fromIndex) Method

Description

This method returns the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.

Syntax

Here is the syntax of this method:

```
public int lastIndexOf(int ch, int fromIndex)
```

Parameters

Here is the detail of parameters:

- **ch** -- a character.

- **fromIndex** -- the index to start the search from.

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

This will produce the following result:

```
Found Last Index :4
```

Java – String lastIndexOf(String str) Method

Description

This method accepts a String as an argument, if the string argument occurs one or more times as a substring within this object, then it returns the index of the first character of the last such substring is returned. If it does not occur as a substring, -1 is returned.

Syntax

Here is the syntax of this method:

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

Parameters

Here is the detail of parameters:

- **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" );
        System.out.print("Found Last Index :" );
        System.out.println( Str.lastIndexOf( SubStr1 ));
    }
}
```

This will produce the following result:

```
Found Last Index :11
```

Java – String lastIndexOf(String str, int fromIndex) Method

Description

This method 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:

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

Parameters

Here is the detail of parameters:

- **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" );
        System.out.print("Found Last Index :" );
        System.out.println( Str.lastIndexOf( SubStr1, 15 ));
    }
}
```

This will produce the following result:

```
Found Last Index :11
```

Java – String length() Method

Description

This method returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

Syntax

Here is the syntax of this method:

```
public int length()
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- This method returns the the length of the sequence of characters represented by this object.

Example

```
import java.io.*;

public class Test{
    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");
        String Str2 = new String("Tutorials" );

        System.out.print("String Length : " );
        System.out.println(Str1.length());

        System.out.print("String Length : " );
        System.out.println(Str2.length());
    }
}
```

This will produce the following result:

```
String Length :29
String Length :9
```

Java – String matches() Method

Description

This method tells whether or not this string matches the given regular expression. An invocation of this method of the form `str.matches(regex)` yields exactly the same result as the expression `Pattern.matches(regex, str)`.

Syntax

Here is the syntax of this method:

```
public boolean matches(String regex)
```

Parameters

Here is the detail of parameters:

- **regex** -- the regular expression to which this string is to be matched.

Return Value

- This method returns true if, and only if, this string matches the given regular expression.

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.matches("(.*).Tutorials(.*)"));

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

        System.out.print("Return Value :" );
        System.out.println(Str.matches("Welcome(.*)"));
    }
}
```

This will produce the following result:

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

Java – String regionMatches() Method

Description

This method has two variants which can be used to test if two string regions are equal.

Syntax

Here is the syntax of this method:

```
public boolean regionMatches(int toffset,
                             String other,
                             int ooffset,
```

```

                                int len)

or

public boolean regionMatches(boolean ignoreCase,
                                int toffset,
                                String other,
                                int ooffset,
                                int len)

```

Parameters

Here is the detail of parameters:

- **toffset** -- the starting offset of the subregion in this string.
- **other** -- the string argument.
- **ooffset** -- the starting offset of the subregion in the string argument.
- **len** -- the number of characters to compare.
- **ignoreCase** -- if true, ignore case when comparing characters.

Return Value

- It returns true if the specified subregion of this string matches the specified subregion of the string argument; false otherwise. Whether the matching is exact or case insensitive depends on the ignoreCase argument.

Example

```

import java.io.*;

public class Test{
    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");
        String Str2 = new String("Tutorials");
        String Str3 = new String("TUTORIALS");

        System.out.print("Return Value : " );
        System.out.println(Str1.regionMatches(11, Str2, 0, 9));

        System.out.print("Return Value : " );
    }
}

```

```
        System.out.println(Str1.regionMatches(11, Str3, 0, 9));

        System.out.print("Return Value : " );
        System.out.println(Str1.regionMatches(true, 11, Str3, 0, 9));
    }
}
```

This will produce the following result:

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

Java – String regionMatches() Method

Description

This method has two variants which can be used to test if two string regions are equal.

Syntax

Here is the syntax of this method:

```
public boolean regionMatches(int toffset,
                             String other,
                             int ooffset,
                             int len)

or

public boolean regionMatches(boolean ignoreCase,
                             int toffset,
                             String other,
                             int ooffset,
                             int len)
```

Parameters

Here is the detail of parameters:

- **toffset** -- the starting offset of the subregion in this string.
- **other** -- the string argument.
- **ooffset** -- the starting offset of the subregion in the string argument.
- **len** -- the number of characters to compare.
- **ignoreCase** -- if true, ignore case when comparing characters.

Return Value

- It returns true if the specified subregion of this string matches the specified subregion of the string argument; false otherwise. Whether the matching is exact or case insensitive depends on the ignoreCase argument.

Example

```
import java.io.*;

public class Test{
    public static void main(String args[]){
        String Str1 = new String("Welcome to Tutorialspoint.com");
        String Str2 = new String("Tutorials");
        String Str3 = new String("TUTORIALS");

        System.out.print("Return Value :" );
        System.out.println(Str1.regionMatches(11, Str2, 0, 9));

        System.out.print("Return Value :" );
        System.out.println(Str1.regionMatches(11, Str3, 0, 9));

        System.out.print("Return Value :" );
        System.out.println(Str1.regionMatches(true, 11, Str3, 0, 9));
    }
}
```

This will produce the following result:

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

Java – String replace() Method

Description

This method returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

Syntax

Here is the syntax of this method:

```
public String replace(char oldChar, char newChar)
```

Parameters

Here is the detail of parameters:

- **oldChar** -- the old character.
- **newChar** -- the new character.

Return Value

- It returns a string derived from this string by replacing every occurrence of oldChar with newChar.

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.replace('o', 'T'));

        System.out.print("Return Value :" );
        System.out.println(Str.replace('l', 'D'));
    }
}
```

This will produce the following result:

```
Return Value :WelcTme tT TutTrialspTint.cTm
Return Value :WeDcome to TutoriaDspoint.com
```


Java – String replaceAll() Method

Description

This method replaces each substring of this string that matches the given regular expression with the given replacement.

Syntax

Here is the syntax of this method:

```
public String replaceAll(String regex, String replacement)
```

Parameters

Here is the detail of parameters:

- **regex** -- the regular expression to which this string is to be matched.
- **replacement** -- the string which would replace found expression.

Return Value

- This method returns the resulting String.

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.replaceAll("(.*?)Tutorials(.*?)",
                                         "AMROOD" ));
    }
}
```

This will produce the following result:

```
Return Value :AMROOD
```

Java – String replaceFirst() Method

Description

This method replaces the first substring of this string that matches the given regular expression with the given replacement.

Syntax

Here is the syntax of this method:

```
public String replaceFirst(String regex, String replacement)
```

Parameters

Here is the detail of parameters:

- **regex** -- the regular expression to which this string is to be matched.
- **replacement** -- the string which would replace found expression.

Return Value

- This method returns a resulting String.

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.replaceFirst("(.*?)Tutorial(.*?)",
            "AMROOD" ));

        System.out.print("Return Value :" );
        System.out.println(Str.replaceFirst("Tutorial", "AMROOD" ));
    }
}
```

This will produce the following result:

```
Return Value :AMROOD
Return Value :Welcome to AMROODpoint.com
```

Java – String split() Method

Description

This method has two variants and splits this string around matches of the given regular expression.

Syntax

Here is the syntax of this method:

```
public String[] split(String regex, int limit)

or

public String[] split(String regex)
```

Parameters

Here is the detail of parameters:

- **regex** -- the delimiting regular expression.
- **limit** -- the result threshold, which means how many strings to be returned.

Return Value

- It returns the array of strings computed by splitting this string around matches of the given regular expression.

Example

```
import java.io.*;

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

        System.out.println("Return Value :" );
        for (String retval: Str.split("-", 2)){
            System.out.println(retval);
        }
        System.out.println("");
        System.out.println("Return Value :" );
        for (String retval: Str.split("-", 3)){
```

```
        System.out.println(retval);
    }
    System.out.println("");
    System.out.println("Return Value : " );
    for (String retval: Str.split("-", 0)){
        System.out.println(retval);
    }
    System.out.println("");
    System.out.println("Return Value : " );
    for (String retval: Str.split("-")){
        System.out.println(retval);
    }
}
}
```

This will produce 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
```

Java – String split() Method

Description

This method has two variants and splits this string around matches of the given regular expression.

Syntax

Here is the syntax of this method:

```
public String[] split(String regex, int limit)

or

public String[] split(String regex)
```

Parameters

Here is the detail of parameters:

- **regex** -- the delimiting regular expression.
- **limit** -- the result threshold which means how many strings to be returned.

Return Value

- It returns the array of strings computed by splitting this string around matches of the given regular expression.

Example

```
import java.io.*;

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

        System.out.println("Return Value :" );
        for (String retval: Str.split("-", 2)){
            System.out.println(retval);
        }
        System.out.println("");
        System.out.println("Return Value :" );
        for (String retval: Str.split("-", 3)){
```

```
        System.out.println(retval);
    }
    System.out.println("");
    System.out.println("Return Value : " );
    for (String retval: Str.split("-", 0)){
        System.out.println(retval);
    }
    System.out.println("");
    System.out.println("Return Value : " );
    for (String retval: Str.split("-")){
        System.out.println(retval);
    }
}
}
```

This will produce 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
```

Java – String startsWith() Method

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.

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 will produce the following result:

```

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

```

Java – String startsWith() Method

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.

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 will produce the following result:

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

Java – String subsequence() Method

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

- This method returns the specified subsequence.

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.subSequence(0, 10) );

        System.out.print("Return Value : " );
        System.out.println(Str.subSequence(10, 15) );
    }
}
```

This will produce the following result:

```
Return Value :Welcome to
Return Value : Tuto
```

Java – String substring() Method

Description

This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex – 1, if the second argument is given.

Syntax

Here is the syntax of this method:

```
public String substring(int beginIndex)

or

public String substring(int beginIndex, int endIndex)
```

Parameters

Here is the detail of parameters:

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

Return Value

- The specified substring.

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.substring(10) );

        System.out.print("Return Value :" );
        System.out.println(Str.substring(10, 15) );
    }
}
```

This will produce the following result:

```
Return Value : Tutorialspoint.com
Return Value : Tuto
```

Java – String substring() Method

Description

This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex - 1, if the second argument is given.

Syntax

Here is the syntax of this method:

```
public String substring(int beginIndex)

or

public String substring(int beginIndex, int endIndex)
```

Parameters

Here is the detail of parameters:

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

Return Value

- The specified substring.

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.substring(10) );

        System.out.print("Return Value :" );
        System.out.println(Str.substring(10, 15) );
    }
}
```

This will produce the following result:

```
Return Value : Tutorialspoint.com
Return Value : Tuto
```

Java – String toCharArray() Method

Description

This method converts this string to a new character array.

Syntax

Here is the syntax of this method:

```
public char[] toCharArray()
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- It returns a newly allocated character array, whose length is the length of this string and whose contents are initialized to contain the character sequence represented by this string.

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.toCharArray() );
    }
}
```

This will produce the following result:

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

Java – String toLowerCase() Method

Description

This method has two variants. The 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())`.

The 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 will produce the following result:

```
Return Value :welcome to tutorialspoint.com
```

Java – String toLowerCase() Method

Description

This method has two variants. The 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()).

The 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 will produce the following result:

```
Return Value :welcome to tutorialspoint.com
```

Java – String toString() Method

Description

This method returns itself a string.

Syntax

Here is the syntax of this method:

```
public String toString()
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- This method returns the string itself.

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.toString());
    }
}
```

This will produce the following result:

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

Java – String toUpperCase() Method

This method has two variants. The first variant converts all of the characters in this String to upper case using the rules of the given Locale. This is equivalent to calling toUpperCase(Locale.getDefault()).

The second variant takes locale as an argument to be used while converting into upper case.

Syntax

Here is the syntax of this method:

```
public String toUpperCase()

or

public String toUpperCase(Locale locale)
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- It returns the String, converted to uppercase.

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.toUpperCase() );
    }
}
```

This will produce the following result:

```
Return Value :WELCOME TO TUTORIALSPOINT.COM
```

Java – String toUpperCase() Method

This method has two variants. The first variant converts all of the characters in this String to upper case using the rules of the given Locale. This is equivalent to calling `toUpperCase(Locale.getDefault())`.

The second variant takes locale as an argument to be used while converting into upper case.

Syntax

Here is the syntax of this method:

```
public String toUpperCase()

or

public String toUpperCase(Locale locale)
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- It returns the String, converted to uppercase.

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.toUpperCase() );
    }
}
```

This produces the following result:

```
Return Value :WELCOME TO TUTORIALSPOINT.COM
```

Java – String trim() Method

Description

This method returns a copy of the string, with leading and trailing whitespace omitted.

Syntax

Here is the syntax of this method:

```
public String trim()
```

Parameters

Here is the detail of parameters:

- NA

Return Value

- 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
```

Java – String valueOf() Method

Description

This method has the following 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

static String valueOf(char[] data, int offset, int count)

or

static String valueOf(double d)

or

static String valueOf(float f)

or

static String valueOf(int i)
```

or

```
static String valueOf(long l)
```

or

```
static String valueOf(Object obj)
```

Parameters

Here is the detail of parameters:

- See the description.

Return Value

- This method returns the string representation.

Example

```
import java.io.*;

public class Test{
    public static void main(String args[]){
        double d = 102939939.939;
        boolean b = true;
        long l = 1232874;

        char[] arr = {'a', 'b', 'c', 'd', 'e', 'f','g' };

        System.out.println("Return Value : " + String.valueOf(d) );
        System.out.println("Return Value : " + String.valueOf(b) );
        System.out.println("Return Value : " + String.valueOf(l) );
        System.out.println("Return Value : " + String.valueOf(arr) );
    }
}
```

This will produce the following result:

```
Return Value : 1.02939939939E8
```

```
Return Value : true
```

```
Return Value : 1232874
```

```
Return Value : abcdefg
```

14. Java – Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar;    // preferred way.  
  
or  
  
dataType arrayRefVar[];    // works but not preferred way.
```

Note: The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax:

```
double[] myList;           // preferred way.  
  
or  
  
double myList[];          // works but not preferred way.
```

Creating Arrays

You can create an array by using the `new` operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using `new dataType[arraySize]`.
- It assigns the reference of the newly created array to the variable `arrayRefVar`.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

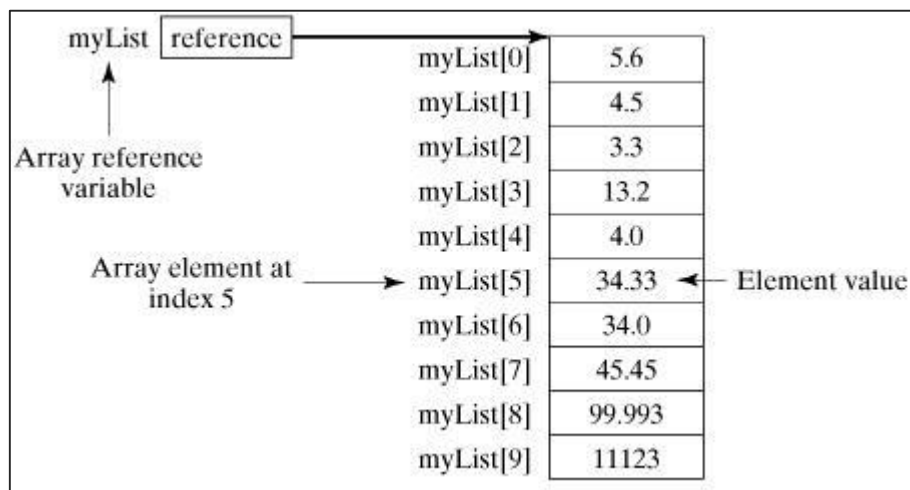
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **`arrayRefVar.length-1`**.

Example

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

```
double[] myList = new double[10];
```

Following picture represents array `myList`. Here, `myList` holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays:

```
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);
    }
}
```

This will produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

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 will 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) {  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
}
```

You can invoke it by passing an array. For example, the following statement invokes the `printArray` method to display 3, 1, 2, 6, 4, and 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array:

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];

    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
        result[j] = list[i];
    }
    return result;
}
```

The Arrays Class

The `java.util.Arrays` class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

Sr. No.	Methods with Description
1	<p>public static int binarySearch(Object[] a, Object key)</p> <p>Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns (- (insertion point + 1)).</p>
2	<p>public static boolean equals(long[] a, long[] a2)</p> <p>Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)</p>

3	public static void fill(int[] a, int val) Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)
4	public static void sort(Object[] a) Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

15. Java – Date & 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 as shown in the following table.

Sr.No.	Constructor and Description
1	Date() This constructor initializes the object with the current date and time.
2	Date(long millisec) This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Following are the methods of the date class.

Sr.No.	Methods with Description
1	boolean after(Date date) 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) 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() Duplicates the invoking Date object.
4	int compareTo(Date date) 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) Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException.
6	boolean equals(Object date) 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() Returns the number of milliseconds that have elapsed since January 1, 1970.
8	int hashCode() Returns a hash code for the invoking object.
9	void setTime(long time) Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970
10	String toString() Converts the invoking Date object into a string and returns the result.

Getting Current Date & Time

This is a very easy method to get current date and time in Java. You can use a simple Date object with *toString()* method to print the current date and time as follows:

```
import java.util.Date;

public class DateDemo {
    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date using toString()
        System.out.println(date.toString());
    }
}
```

This will produce the following result:

```
on May 04 09:51:52 CDT 2009
```

Date Comparison

Following are the three ways to compare two dates:

- You can use `getTime()` to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
- You can use the methods `before()`, `after()`, and `equals()`. Because the 12th of the month comes before the 18th, for example, `new Date(99, 2, 12).before(new Date(99, 2, 18))` returns true.
- You can use the `compareTo()` method, which is defined by the `Comparable` interface and implemented by `Date`.

Date Formatting Using SimpleDateFormat

`SimpleDateFormat` is a concrete class for formatting and parsing dates in a locale-sensitive manner. `SimpleDateFormat` allows you to start by choosing any user-defined patterns for date-time formatting. For example:

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

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

        Date dNow = new Date( );
        SimpleDateFormat ft =
            new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

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

This will produce the following result:

```
Current Date: Sun 2004.07.18 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 as shown in the following code. For example:

```
import java.util.Date;

public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date using toString()
        String str = String.format("Current Date/Time : %tc", date );

        System.out.printf(str);
    }
}
```

This will produce the following result:

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the index of the argument to be formatted.

The index must immediately follow the % and it must be terminated by a \$. For example:

```
import java.util.Date;

public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();
```

```

        // display time and date using toString()
        System.out.printf("%1$s %2$tB %2$td, %2$tY",
                           "Due date:", date);
    }
}

```

This will produce the following result:

```
Due date: February 09, 2004
```

Alternatively, you can use the < flag. It indicates that the same argument as in the preceding format specification should be used again. For example:

```

import java.util.Date;

public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display formatted date
        System.out.printf("%s %tB %<te, %<tY",
                           "Due date:", date);
    }
}

```

This will produce the following result:

```
Due date: February 09, 2004
```

Date and Time Conversion Characters

Character	Description	Example
c	Complete date and time	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 date	2004-02-09
D	U.S. formatted date (month/day/year)	02/09/2004
T	24-hour time	18:05:19
r	12-hour time	06:05:19 pm
R	24-hour time, no seconds	18:05
Y	Four-digit year (with leading zeroes)	2004
y	Last two digits of the year (with leading zeroes)	04
C	First two digits of the year (with leading zeroes)	20
B	Full month name	February
b	Abbreviated month name	Feb
m	Two-digit month (with leading zeroes)	02
d	Two-digit day (with leading zeroes)	03
e	Two-digit day (without leading zeroes)	9
A	Full weekday name	Monday
a	Abbreviated weekday name	Mon
j	Three-digit day of year (with leading zeroes)	069
H	Two-digit hour (with leading zeroes), between 00 and 23	18
k	Two-digit hour (without leading zeroes), between 0 and 23	18

I	Two-digit hour (with leading zeroes), between 01 and 12	06
l	Two-digit hour (without leading zeroes), between 1 and 12	6
M	Two-digit minutes (with leading zeroes)	05
S	Two-digit seconds (with leading zeroes)	19
L	Three-digit milliseconds (with leading zeroes)	047
N	Nine-digit nanoseconds (with leading zeroes)	047000000
P	Uppercase morning or afternoon marker	PM
p	Lowercase morning or afternoon marker	pm
z	RFC 822 numeric offset from GMT	-0800
Z	Time zone	PST
s	Seconds since 1970-01-01 00:00:00 GMT	1078884319
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
1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818
$ java DateDemo 2007-12-01
2007-12-01 Parses as Sat Dec 01 00:00:00 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, the 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!");
        }
    }
}

```

This will produce the following result:

```
Sun May 03 18:04:41 GMT 2009
```

```
Sun May 03 18:04:51 GMT 2009
```

Measuring Elapsed Time

Sometimes, you may need to measure point in time in milliseconds. So let's re-write the above example once again:

```

import java.util.*;

public class DiffDemo {

    public static void main(String args[]) {
        try {
            long start = System.currentTimeMillis( );
            System.out.println(new Date( ) + "\n");
            Thread.sleep(5*60*10);
            System.out.println(new Date( ) + "\n");
            long end = System.currentTimeMillis( );
            long diff = end - start;
            System.out.println("Difference is : " + diff);
        } catch (Exception e) {
            System.out.println("Got an exception!");
        }
    }
}

```

This will produce the following result:

```
Sun May 03 18:16:51 GMT 2009
```

```
Sun May 03 18:16:57 GMT 2009
```

```
Difference is : 5993
```

GregorianCalendar Class

GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. We did not discuss Calendar class in this tutorial, you can look up standard Java documentation for this.

The **getInstance()** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for GregorianCalendar objects:

Sr. No.	Constructor with Description
1	GregorianCalendar() Constructs a default GregorianCalendar using the current time in the default time zone with the default locale.
2	GregorianCalendar(int year, int month, int date) Constructs a GregorianCalendar with the given date set in the default time zone with the default locale.
3	GregorianCalendar(int year, int month, int date, int hour, int minute) Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.
4	GregorianCalendar(int year, int month, int date, int hour, int minute, int second) Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.

5	GregorianCalendar(Locale aLocale) Constructs a GregorianCalendar based on the current time in the default time zone with the given locale.
6	GregorianCalendar(TimeZone zone) Constructs a GregorianCalendar based on the current time in the given time zone with the default locale.
7	GregorianCalendar(TimeZone zone, Locale aLocale) Constructs a GregorianCalendar based on the current time in the given time zone with the given locale.

Here is the list of few useful support methods provided by GregorianCalendar class:

Sr. No.	Methods with Description
1	void add(int field, int amount) Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.
2	protected void computeFields() Converts UTC as milliseconds to time field values.
3	protected void computeTime() Overrides Calendar Converts time field values to UTC as milliseconds.
4	boolean equals(Object obj) Compares this GregorianCalendar to an object reference.
5	int get(int field) Gets the value for a given time field.

6	int getActualMaximum(int field) Returns the maximum value that this field could have, given the current date.
7	int getActualMinimum(int field) Returns the minimum value that this field could have, given the current date.
8	int getGreatestMinimum(int field) Returns highest minimum value for the given field if varies.
9	Date getGregorianChange() Gets the Gregorian Calendar change date.
10	int getLeastMaximum(int field) Returns lowest maximum value for the given field if varies.
11	int getMaximum(int field) Returns maximum value for the given field.
12	Date getTime() Gets this Calendar's current time.
13	long getTimeInMillis() Gets this Calendar's current time as a long.
14	TimeZone getTimeZone() Gets the time zone.
15	int getMinimum(int field) Returns minimum value for the given field.
16	int hashCode() Overrides hashCode.
17	boolean isLeapYear(int year) Determines if the given year is a leap year.

18	void roll(int field, boolean up) Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields.
19	void set(int field, int value) Sets the time field with the given value.
20	void set(int year, int month, int date) Sets the values for the fields year, month, and date.
21	void set(int year, int month, int date, int hour, int minute) Sets the values for the fields year, month, date, hour, and minute.
22	void set(int year, int month, int date, int hour, int minute, int second) Sets the values for the fields year, month, date, hour, minute, and second.
23	void setGregorianCalendar(Date date) Sets the GregorianCalendar change date.
24	void setTime(Date date) Sets this Calendar's current time with the given Date.
25	void setTimeInMillis(long millis) Sets this Calendar's current time from the given long value.
26	void setTimeZone(TimeZone value) Sets the time zone with the given time zone value.
27	String toString() Returns a string representation of this calendar.

Example

```
import java.util.*;

public class GregorianCalendarDemo {

    public static void main(String args[]) {
        String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

        int year;
        // Create a Gregorian calendar initialized
        // with the current date and time in the
        // default locale and timezone.
        GregorianCalendar gcalendar = new GregorianCalendar();
        // Display current time and date information.
        System.out.print("Date: ");
        System.out.print(months[gcalendar.get(Calendar.MONTH)]);
        System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
        System.out.println(year = gcalendar.get(Calendar.YEAR));
        System.out.print("Time: ");
        System.out.print(gcalendar.get(Calendar.HOUR) + ":");
        System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
        System.out.println(gcalendar.get(Calendar.SECOND));

        // Test if the current year is a leap year
        if(gcalendar.isLeapYear(year)) {
            System.out.println("The current year is a leap year");
        }
        else {
            System.out.println("The current year is not a leap year");
        }
    }
}
```

This will produce the following result:

```
Date: Apr 22 2009  
Time: 11:25:27  
The current year is not a leap year
```

For a complete list of constant available in Calendar class, you can refer the standard Java documentation.

16. Java – Regular Expressions

Java provides the `java.util.regex` package for pattern matching with regular expressions. Java regular expressions are very similar to the Perl programming language and very easy to learn.

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The `java.util.regex` package primarily consists of the following three classes:

- **Pattern Class:** A `Pattern` object is a compiled representation of a regular expression. The `Pattern` class provides no public constructors. To create a pattern, you must first invoke one of its public static **`compile()`** methods, which will then return a `Pattern` object. These methods accept a regular expression as the first argument.
- **Matcher Class:** A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string. Like the `Pattern` class, `Matcher` defines no public constructors. You obtain a `Matcher` object by invoking the **`matcher()`** method on a `Pattern` object.
- **PatternSyntaxException:** A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Capturing Groups

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression `(dog)` creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from the left to the right. In the expression `((A)(B(C)))`, for example, there are four such groups:

- `((A)(B(C)))`
- `(A)`
- `(B(C))`
- `(C)`

To find out how many groups are present in the expression, call the `groupCount` method on a `matcher` object. The `groupCount` method returns an **`int`** showing the number of capturing groups present in the `matcher`'s pattern.

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 placed 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 will produce the following result:

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed 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
<code>^</code>	Matches the beginning of the line.
<code>\$</code>	Matches the end of the line.
<code>.</code>	Matches any single character except newline. Using m option allows it to match the newline as well.
<code>[...]</code>	Matches any single character in brackets.
<code>[^...]</code>	Matches any single character not in brackets.
<code>\A</code>	Beginning of the entire string.
<code>\Z</code>	End of the entire string.
<code>\Z</code>	End of the entire string except allowable final line terminator.
<code>re*</code>	Matches 0 or more occurrences of the preceding expression.
<code>re+</code>	Matches 1 or more of the previous thing.

<code>re?</code>	Matches 0 or 1 occurrence of the preceding expression.
<code>re{ n}</code>	Matches exactly n number of occurrences of the preceding expression.
<code>re{ n,}</code>	Matches n or more occurrences of the preceding expression.
<code>re{ n, m}</code>	Matches at least n and at most m occurrences of the preceding expression.
<code>a b</code>	Matches either a or b.
<code>(re)</code>	Groups regular expressions and remembers the matched text.
<code>(?: re)</code>	Groups regular expressions without remembering the matched text.
<code>(?> re)</code>	Matches the independent pattern without backtracking.
<code>\w</code>	Matches the word characters.
<code>\W</code>	Matches the nonword characters.
<code>\s</code>	Matches the whitespace. Equivalent to <code>[\t\n\r\f]</code> .
<code>\S</code>	Matches the nonwhitespace.

<code>\d</code>	Matches the digits. Equivalent to <code>[0-9]</code> .
<code>\D</code>	Matches the nondigits.
<code>\A</code>	Matches the beginning of the string.
<code>\Z</code>	Matches the end of the string. If a newline exists, it matches just before newline.
<code>\z</code>	Matches the end of the string.
<code>\G</code>	Matches the point where the last match finished.
<code>\n</code>	Back-reference to capture group number "n".
<code>\b</code>	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
<code>\B</code>	Matches the nonword boundaries.
<code>\n, \t, etc.</code>	Matches newlines, carriage returns, tabs, etc.
<code>\Q</code>	Escape (quote) all characters up to <code>\E</code> .
<code>\E</code>	Ends quoting begun with <code>\Q</code> .

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:

Sr. No.	Methods with Description
1	public int start() Returns the start index of the previous match.
2	public int start(int group) Returns the start index of the subsequence captured by the given group during the previous match operation.
3	public int end() Returns the offset after the last character matched.
4	public int end(int group) Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Study Methods

Study methods review the input string and return a Boolean indicating whether or not the pattern is found:

Sr. No.	Methods with Description
1	public boolean lookingAt() Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2	public boolean find() Attempts to find the next subsequence of the input sequence that matches the pattern.

3	public boolean find(int start) Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4	public boolean matches() Attempts to match the entire region against the pattern.

Replacement Methods

Replacement methods are useful methods for replacing text in an input string:

Sr. No.	Methods with Description
1	public Matcher appendReplacement(StringBuffer sb, String replacement) Implements a non-terminal append-and-replace step.
2	public StringBuffer appendTail(StringBuffer sb) Implements a terminal append-and-replace step.
3	public String replaceAll(String replacement) Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4	public String replaceFirst(String replacement) Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5	public static String quoteReplacement(String s) Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement s in the appendReplacement method of the Matcher class.

The *start* and *end* Methods

Following is the example that counts the number of times the word "cat" appears in the input string:

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

public class RegexMatches
{
    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT =
        "cat cat cat cattie cat";

    public static void main( String args[] ){
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT); // get a matcher object
        int count = 0;

        while(m.find()) {
            count++;
            System.out.println("Match number "+count);
            System.out.println("start(): "+m.start());
            System.out.println("end(): "+m.end());
        }
    }
}
```

This will produce the following result:

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
```

```

Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22

```

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and the end returns the index of the last character matched, plus one.

The *matches* and *lookingAt* Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality:

```

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

public class RegexMatches
{
    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main( String args[] ){
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
        System.out.println("Current INPUT is: "+INPUT);
    }
}

```

```

        System.out.println("lookingAt(): "+matcher.lookingAt());
        System.out.println("matches(): "+matcher.matches());
    }
}

```

This will produce the following result:

```

Current REGEX is: foo
Current INPUT is: fooooooooooooooooooooo
lookingAt(): true
matches(): false

```

The replaceFirst and replaceAll Methods

The replaceFirst and replaceAll methods replace the text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.

Here is the example explaining the functionality:

```

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

public class RegexMatches
{
    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " +
                                   "All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}

```

This will produce the following result:

```
The cat says meow. All cats say meow.
```

The appendReplacement and appendTail Methods

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality:

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

public class RegexMatches
{
    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";
    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        StringBuffer sb = new StringBuffer();
        while(m.find()){
            m.appendReplacement(sb, REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

This will produce the following result:

```
-foo-foo-foo-
```

PatternSyntaxException Class Methods

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong:

Sr. No.	Methods with Description
1	public String getDescription() Retrieves the description of the error.
2	public int getIndex() Retrieves the error index.
3	public String getPattern() Retrieves the erroneous regular expression pattern.
4	public String getMessage() Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern.

17. Java – Methods

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

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating Method

Considering the following example to explain the syntax of a method:

```
public static int methodName(int a, int b) {  
    // body  
}
```

Here,

- **public static:** modifier
- **int:** return type
- **methodName:** name of the method
- **a, b:** formal parameters
- **int a, int b:** list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax:

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

The syntax shown above includes:

- **modifier:** It defines the access type of the method and it is optional to use.
- **returnType:** Method may return a value.
- **nameOfMethod:** This is the method name. The method signature consists of the method name and the parameter list.

- **Parameter List:** The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body:** The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **max()**. This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
```

Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example:

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

Example

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber{

    public static void main(String[] args) {

        int a = 11;
        int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }

    /** returns the minimum of two numbers */
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
}
```

This will produce the following result:

```
Minimum value = 6
```

The void Keyword

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown in the following example.

Example

```
public class ExampleVoid {  
  
    public static void main(String[] args) {  
        methodRankPoints(255.7);  
    }  
  
    public static void methodRankPoints(double points) {  
        if (points >= 202.5) {  
            System.out.println("Rank:A1");  
        }  
        else if (points >= 122.4) {  
            System.out.println("Rank:A2");  
        }  
        else {  
            System.out.println("Rank:A3");  
        }  
    }  
}
```

This will produce the following result:

```
Rank:A1
```

Passing Parameters by Value

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

Example

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```
public class swappingExample {

    public static void main(String[] args) {

        int a = 30;
        int b = 45;

        System.out.println("Before swapping, a = " +
                           a + " and b = " + b);

        // Invoke the swap method
        swapFunction(a, b);
        System.out.println("\n**Now, Before and After swapping values will be
same here**");
        System.out.println("After swapping, a = " +
                           a + " and b is " + b);
    }

    public static void swapFunction(int a, int b) {

        System.out.println("Before swapping(Inside), a = " + a
                           + " b = " + b);

        // Swap n1 with n2
        int c = a;
        a = b;
        b = c;

        System.out.println("After swapping(Inside), a = " + a
                           + " b = " + b);
    }
}
```

This will produce the following result:

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same:

```
public class ExampleOverloading{

    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        double c = 7.3;
        double d = 9.4;
        int result1 = minFunction(a, b);
        // same function name with different parameters
        double result2 = minFunction(c, d);
        System.out.println("Minimum Value = " + result1);
        System.out.println("Minimum Value = " + result2);
    }

    // for integer
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
```

```
        min = n1;

    return min;
}
// for double
public static double minFunction(double n1, double n2) {
    double min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
}
```

This will produce the following result:

```
Minimum Value = 6
Minimum Value = 7.3
```

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

Using Command-Line Arguments

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to `main()`.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the `String` array passed to `main()`.

Example

The following program displays all of the command-line arguments that it is called with:

```
public class CommandLine {  
  
    public static void main(String args[]){  
        for(int i=0; i<args.length; i++){  
            System.out.println("args[" + i + "]: " +  
                               args[i]);  
        }  
    }  
}
```

Try executing this program as shown here:

```
$java CommandLine this is a command line 200 -100
```

This will produce the following result:

```
args[0]: this  
args[1]: is  
args[2]: a  
args[3]: command  
args[4]: line  
args[5]: 200  
args[6]: -100
```

The Constructors

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example

Here is a simple example that uses a constructor without parameters:

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass() {
        x = 10;
    }
}
```

You will have to call constructor to initialize objects as follows:

```
public class ConsDemo {

    public static void main(String args[]) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();
        System.out.println(t1.x + " " + t2.x);
    }
}
```

Parameterized Constructor

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor with a parameter:

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}
```

```

    }
}

```

You will need to call a constructor to initialize objects as follows:

```

public class ConsDemo {

    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}

```

This will produce the following result:

```

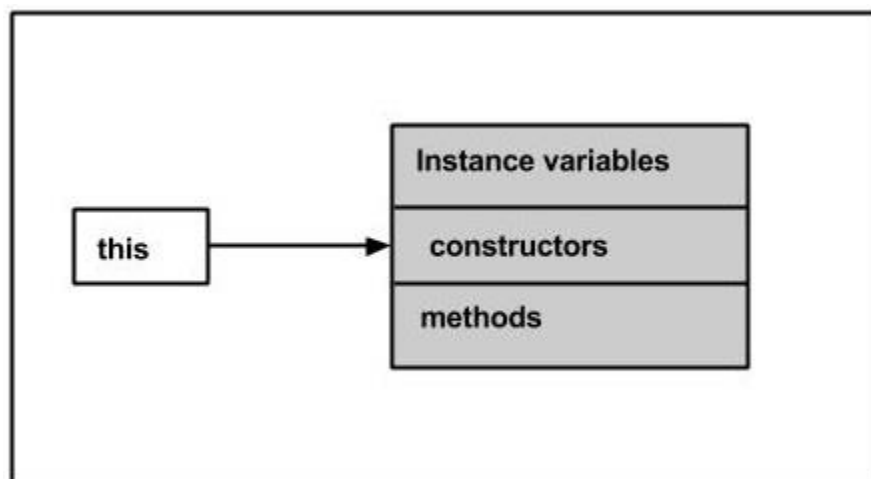
10 20

```

The this keyword

this is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

Note: The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to :

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student{

    int age;
    Student(int age){
        this.age=age;
    }

}
```

- Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

```
class Student{

    int age
    Student(){
        this(20);
    }

    Student(int age){
        this.age=age;
    }

}
```

Example

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the following program in a file with the name, **This_Example.java**.

```
public class This_Example {

    //Instance variable num
    int num=10;

    This_Example(){
        System.out.println("This is an example program on keyword this ");
    }

    This_Example(int num){
        //Invoking the default constructor
    }

}
```

```
this();

//Assigning the local variable num to the instance variable num
this.num=num;
}

public void greet(){
    System.out.println("Hi Welcome to Tutorialspoint");
}

public void print(){
    //Local variable num
    int num=20;

    //Printing the instance variable
    System.out.println("value of local variable num is : "+num);

    //Printing the local variable
    System.out.println("value of instance variable num is : "+this.num);

    //Invoking the greet method of a class
    this.greet();
}

public static void main(String[] args){
    //Instantiating the class
    This_Example obj1=new This_Example();

    //Invoking the print method
    obj1.print();

    //Passing a new value to the num variable through parametrized constructor
    This_Example obj2=new This_Example(30);

    //Invoking the print method again
    obj2.print();
}
}
```

This will produce the following result:

```

This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 10
Hi Welcome to Tutorialspoint
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 30
Hi Welcome to Tutorialspoint

```

Variable Arguments(var-args)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example

```

public class VarargsDemo {

    public static void main(String args[]) {

        // Call method with variable args
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }

    public static void printMax( double... numbers) {
    if (numbers.length == 0) {
        System.out.println("No argument passed");
        return;
    }

    double result = numbers[0];

    for (int i = 1; i < numbers.length; i++)
        if (numbers[i] > result)
            result = numbers[i];
    }
}

```

```
        System.out.println("The max value is " + result);  
    }  
}
```

This will produce the following result:

```
The max value is 56.5  
The max value is 3.0
```

The finalize() Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize()**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize() to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize() method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the finalize() method, you will specify those actions that must be performed before an object is destroyed.

The finalize() method has this general form:

```
protected void finalize( )  
{  
    // finalization code here  
}
```

Here, the keyword protected is a specifier that prevents access to finalize() by code defined outside its class.

This means that you cannot know when or even if finalize() will be executed. For example, if your program ends before garbage collection occurs, finalize() will not execute.

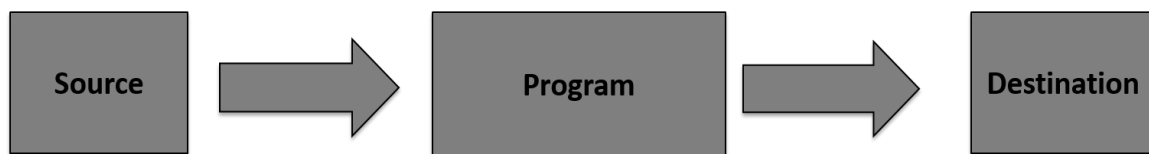
18. Java – Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams:

- **InPutStream:** The InputStream is used to read data from a source.
- **OutPutStream:** The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;

public class CopyFile {
    public static void main(String args[]) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
```

```

        out = new FileOutputStream("output.txt");

        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    }finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
}

```

Now let's have a file **input.txt** with the following content:

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*;

public class CopyFile {
    public static void main(String args[]) throws IOException
    {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Now let's have a file **input.txt** with the following content:

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams:

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q":

```
import java.io.*;

public class ReadConsole {
    public static void main(String args[]) throws IOException
    {
        InputStreamReader cin = null;

        try {
            cin = new InputStreamReader(System.in);
            System.out.println("Enter characters, 'q' to quit.");
            char c;
            do {
                c = (char) cin.read();
                System.out.print(c);
            } while(c != 'q');
        } finally {
            if (cin != null) {
                cin.close();
            }
        }
    }
}
```

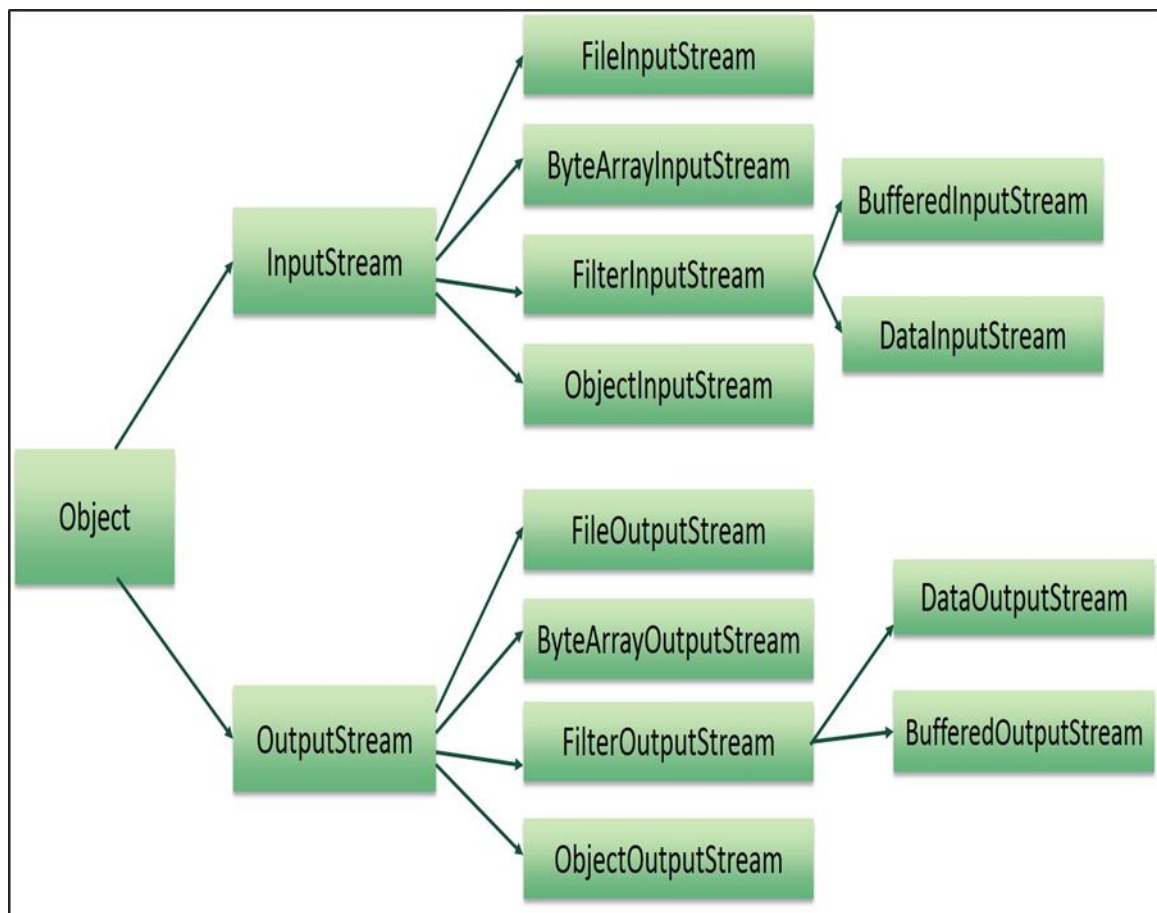
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q':

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr. No.	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.

5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.
---	---

There are other important input streams available, for more detail you can refer to the following links:

- [ByteArrayInputStream](#)
- [DataInputStream](#)

ByteArrayInputStream

The `ByteArrayInputStream` class allows a buffer in the memory to be used as an `InputStream`. The input source is a byte array.

`ByteArrayInputStream` class provides the following constructors.

Sr.No	Constructor and Description
1	ByteArrayInputStream(byte [] a) This constructor accepts a byte array as a parameter.
2	ByteArrayInputStream(byte [] a, int off, int len) This constructor takes an array of bytes, and two integer values, where off is the first byte to be read and len is the number of bytes to be read.

Once you have `ByteArrayInputStream` object in hand then there is a list of helper methods which can be used to read the stream or to do other operations on the stream.

Sr. No.	Methods with Description
1	public int read() This method reads the next byte of data from the <code>InputStream</code> . Returns an int as the next byte of data. If it is the end of the file, then it returns -1.
2	public int read(byte[] r, int off, int len)

	This method reads upto len number of bytes starting from off from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
3	public int available() Gives the number of bytes that can be read from this file input stream. Returns an int that gives the number of bytes to be read.
4	public void mark(int read) This sets the current marked position in the stream. The parameter gives the maximum limit of bytes that can be read before the marked position becomes invalid.
5	public long skip(long n) Skips 'n' number of bytes from the stream. This returns the actual number of bytes skipped.

Example

Following is the example to demonstrate `ByteArrayInputStream` and `ByteArrayOutputStream`.

```
import java.io.*;

public class ByteStreamTest {

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

        ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);

        while( bOutput.size() != 10 ) {
            // Gets the inputs from the user
            bOutput.write(System.in.read());
        }

        byte b [] = bOutput.toByteArray();
        System.out.println("Print the content");
        for(int x= 0 ; x < b.length; x++) {
            // printing the characters

```

```

        System.out.print((char)b[x] + " ");
    }
    System.out.println(" ");

    int c;

    ByteArrayInputStream bInput = new ByteArrayInputStream(b);

    System.out.println("Converting characters to Upper case ");
    for(int y = 0 ; y < 1; y++ ) {
        while(( c= bInput.read())!= -1) {
            System.out.println(Character.toUpperCase((char)c));
        }
        bInput.reset();
    }
}
}

```

Following is the sample run of the above program:

```

asdfghjkly
Print the content
a  s  d  f  g  h  j  k  l  y
Converting characters to Upper case
A
S
D
F
G
H
J
K
L
Y

```

DataInputStream

The `DataInputStream` is used in the context of `DataOutputStream` and can be used to read primitives.

Following is the constructor to create an `InputStream`:

```
InputStream in = DataInputStream(InputStream in);
```

Once you have `DataInputStream` object in hand, then there is a list of helper methods, which can be used to read the stream or to do other operations on the stream.

Sr. No.	Methods with Description
1	public final int read(byte[] r, int off, int len) throws IOException Reads up to len bytes of data from the input stream into an array of bytes. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file.
2	Public final int read(byte [] b) throws IOException Reads some bytes from the inputstream an stores in to the byte array. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file.
3	(a) public final Boolean readBooolean() throws IOException (b) public final byte readByte() throws IOException (c) public final short readShort() throws IOException (d) public final Int readInt() throws IOException These methods will read the bytes from the contained <code>InputStream</code> . Returns the next two bytes of the <code>InputStream</code> as the specific primitive type.
4	public String readLine() throws IOException Reads the next line of text from the input stream. It reads successive bytes, converting each byte separately into a character, until it encounters a line terminator or end of file; the characters read are then returned as a <code>String</code> .

Example

Following is an example to demonstrate `DataInputStream` and `DataOutputStream`. This example reads 5 lines given in a file `test.txt` and converts those lines into capital letters and finally copies them into another file `test1.txt`.

```
import java.io.*;

public class DataInput_Stream{

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

        //writing string to a file encoded as modified UTF-8
        DataOutputStream dataOut = new DataOutputStream(new
        FileOutputStream("E:\\file.txt"));
        dataOut.writeUTF("hello");

        //Reading data from the same file
        DataInputStream dataIn = new DataInputStream(new
        FileInputStream("E:\\file.txt"));

        while(dataIn.available()>0){

            String k = dataIn.readUTF();
            System.out.print(k+" ");
        }

    }

}
```

Following is the sample run of the above program:

```
hello
```

FileOutputStream

`FileOutputStream` is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a `FileOutputStream` object.

Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using `File()` method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr. No.	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an <code>IOException</code> .
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an <code>IOException</code> .
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes <code>w.length</code> bytes from the mentioned byte array to the <code>OutputStream</code> .

There are other important output streams available, for more detail you can refer to the following links:

- [ByteArrayOutputStream](#)
- [DataOutputStream](#)

ByteArrayOutputStream

The `ByteArrayOutputStream` class stream creates a buffer in memory and all the data sent to the stream is stored in the buffer.

Following is the list of the constructors to be provided by `ByteArrayOutputStream` class.

Sr. No.	Constructors and Description
1	ByteArrayOutputStream() This constructor creates a <code>ByteArrayOutputStream</code> having buffer of 32 byte
2	ByteArrayOutputStream(int a) This constructor creates a <code>ByteArrayOutputStream</code> having buffer of the given size

Once you have `ByteArrayOutputStream` object in hand, then there is a list of helper methods which can be used to write the stream or to do other operations on the stream.

Sr. No.	Methods with Description
1	public void reset() This method resets the number of valid bytes of the byte array output stream to zero, so all the accumulated output in the stream will be discarded.
2	public byte[] toByteArray() This method creates a newly allocated Byte array. Its size would be the current size of the output stream and the contents of the buffer will be copied into it. Returns the current contents of the output stream as a byte array.

3	public String toString() Converts the buffer content into a string. Translation will be done according to the default character encoding. Returns the String translated from the buffer's content.
4	public void write(int w) Writes the specified array to the output stream.
5	public void write(byte []b, int of, int len) Writes len number of bytes starting from offset off to the stream.
6	public void writeTo(OutputStream outSt) Writes the entire content of this Stream to the specified stream argument.

Example

Following is an example to demonstrate `ByteArrayOutputStream` and `ByteArrayInputStream`.

```
import java.io.*;

public class ByteStreamTest {

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

        ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);
        while( bOutput.size() != 10 ) {
            // Gets the inputs from the user
            bOutput.write(System.in.read());
        }

        byte b [] = bOutput.toByteArray();
        System.out.println("Print the content");
        for(int x= 0 ; x < b.length; x++) {
            //printing the characters
            System.out.print((char)b[x] + " ");
        }
    }
}
```

```

        System.out.println("    ");

        int c;

        ByteArrayInputStream bInput = new ByteArrayInputStream(b);
        System.out.println("Converting characters to Upper case " );
        for(int y = 0 ; y < 1; y++ ) {
            while(( c= bInput.read())!= -1) {
                System.out.println(Character.toUpperCase((char)c));
            }
            bInput.reset();
        }
    }
}

```

Here is the sample run of the above program:

```

asdfghjkl y
Print the content
a  s  d  f  g  h  j  k  l  y
Converting characters to Upper case
A
S
D
F
G
H
J
K
L
Y

```

DataOutputStream

The `DataOutputStream` stream lets you write the primitives to an output source.

Following is the constructor to create a `DataOutputStream`.

```
DataOutputStream out = new DataOutputStream(OutputStream out);
```

Once you have *DataOutputStream* object in hand, then there is a list of helper methods, which can be used to write the stream or to do other operations on the stream.

Sr. No.	Methods with Description
1	public final void write(byte[] w, int off, int len) throws IOException Writes len bytes from the specified byte array starting at point off, to the underlying stream.
2	Public final int write(byte [] b) throws IOException Writes the current number of bytes written to this data output stream. Returns the total number of bytes written into the buffer.
3	(a) public final void writeBoolean() throws IOException, (b) public final void writeByte() throws IOException, (c) public final void writeShort() throws IOException (d) public final void writeInt() throws IOException These methods will write the specific primitive type data into the output stream as bytes.
4	Public void flush() throws IOException Flushes the data output stream.
5	public final void writeBytes(String s) throws IOException Writes out the string to the underlying output stream as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits.

Example

Following is an example to demonstrate `DataInputStream` and `DataOutputStream`. This example reads 5 lines given in a file `test.txt` and converts those lines into capital letters and finally copies them into another file `test1.txt`.

```
import java.io.*;

public class DataInput_Stream{

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

        //writing string to a file encoded as modified UTF-8
        DataOutputStream dataOut = new DataOutputStream(new
        FileOutputStream("E:\\file.txt"));
        dataOut.writeUTF("hello");

        //Reading data from the same file
        DataInputStream dataIn = new DataInputStream(new
        FileInputStream("E:\\file.txt"));

        while(dataIn.available()>0){

            String k = dataIn.readUTF();
            System.out.print(k+" ");
        }

    }

}
```

Here is the sample run of the above program:

```
THIS IS TEST 1  ,
THIS IS TEST 2  ,
THIS IS TEST 3  ,
THIS IS TEST 4  ,
THIS IS TEST 5  ,
```

Example

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

    public static void main(String args[]){

        try{
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x=0; x < bWrite.length ; x++){
                os.write( bWrite[x] ); // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("test.txt");
            int size = is.available();

            for(int i=0; i< size; i++){
                System.out.print((char)is.read() + " ");
            }
            is.close();
        }catch(IOException e){
            System.out.print("Exception");
        }
    }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- [File Class](#)
- [FileReader Class](#)
- [FileWriter Class](#)

File Class

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc.

The File object represents the actual file/directory on the disk. Following is the list of constructors to create a File object.

Sr. No.	Methods with Description
1	File(File parent, String child) This constructor creates a new File instance from a parent abstract pathname and a child pathname string.
2	File(String pathname) This constructor creates a new File instance by converting the given pathname string into an abstract pathname.
3	File(String parent, String child) This constructor creates a new File instance from a parent pathname string and a child pathname string.
4	File(URI uri) This constructor creates a new File instance by converting the given file: URI into an abstract pathname.

Once you have *File* object in hand, then there is a list of helper methods which can be used to manipulate the files.

Sr. No.	Methods with Description
1	public String getName() Returns the name of the file or directory denoted by this abstract pathname.
2	public String getParent() Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.

3	public File getParentFile() Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4	public String getPath() Converts this abstract pathname into a pathname string.
5	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise.
6	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
7	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
8	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
9	public boolean exists() Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise.
10	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.

11	<p>public boolean isFile()</p> <p>Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.</p>
12	<p>public long lastModified()</p> <p>Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.</p>
13	<p>public long length()</p> <p>Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.</p>
14	<p>public boolean createNewFile() throws IOException</p> <p>Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.</p>
15	<p>public boolean delete()</p> <p>Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.</p>
16	<p>public void deleteOnExit()</p> <p>Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.</p>
17	<p>public String[] list()</p> <p>Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.</p>

18	public String[] list(FilenameFilter filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
20	public File[] listFiles() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
21	public File[] listFiles(FileFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
22	public boolean mkdir() Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise.
23	public boolean mkdirs() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
24	public boolean renameTo(File dest) Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise.
25	public boolean setLastModified(long time) Sets the last-modified time of the file or directory named by this abstract pathname. Returns true if and only if the operation succeeded; false otherwise.
26	public boolean setReadOnly() Marks the file or directory named by this abstract pathname so that only read operations are allowed. Returns true if and only if the operation succeeded; false otherwise.

27	<p>public static File createTempFile(String prefix, String suffix, File directory) throws IOException</p> <p>Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. Returns an abstract pathname denoting a newly-created empty file.</p>
28	<p>public static File createTempFile(String prefix, String suffix) throws IOException</p> <p>Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. Invoking this method is equivalent to invoking createTempFile(prefix, suffix, null). Returns abstract pathname denoting a newly-created empty file.</p>
29	<p>public int compareTo(File pathname)</p> <p>Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.</p>
30	<p>public int compareTo(Object o)</p> <p>Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.</p>
31	<p>public boolean equals(Object obj)</p> <p>Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.</p>
32	<p>public String toString()</p> <p>Returns the pathname string of this abstract pathname. This is just the string returned by the getPath() method.</p>

Example

Following is an example to demonstrate File object:

```
package com.tutorialspoint;

import java.io.File;

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

        File f = null;
        String[] strs = {"test1.txt", "test2.txt"};
        try{
            // for each string in string array
            for(String s:strs )
            {
                // create new file
                f= new File(s);

                // true if the file is executable
                boolean bool = f.canExecute();

                // find the absolute path
                String a = f.getAbsolutePath();

                // prints absolute path
                System.out.print(a);

                // prints
                System.out.println(" is executable: "+ bool);
            }
        }catch(Exception e){
            // if any I/O error occurs
            e.printStackTrace();
        }
    }
}
```

Consider there is an executable file test1.txt and another file test2.txt is non executable in the current directory. Let us compile and run the above program, this will produce the following result:

```
test1.txt is executable: true
test2.txt is executable: false
```

FileReader Class

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.

This class has several constructors to create required objects. Following is the list of constructors provided by the FileReader class.

Sr. No.	Constructors and Description
1	FileReader(File file) This constructor creates a new FileReader, given the File to read from.
2	FileReader(FileDescriptor fd) This constructor creates a new FileReader, given the FileDescriptor to read from.
3	FileReader(String fileName) This constructor creates a new FileReader, given the name of the file to read from.

Once you have FileReader object in hand then there is a list of helper methods which can be used to manipulate the files.

Sr. No.	Methods with Description
1	public int read() throws IOException Reads a single character. Returns an int, which represents the character read.
2	public int read(char [] c, int offset, int len) Reads characters into an array. Returns the number of characters read.

Example

Following is an example to demonstrate class:

```
import java.io.*;

public class FileRead{

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

        File file = new File("Hello1.txt");
        // creates the file
        file.createNewFile();

        // creates a FileWriter Object
        FileWriter writer = new FileWriter(file);

        // Writes the content to the file
        writer.write("This\n is\n an\n example\n");
        writer.flush();
        writer.close();

        //Creates a FileReader Object
        FileReader fr = new FileReader(file);
        char [] a = new char[50];
        fr.read(a); // reads the content to the array
        for(char c : a)
            System.out.print(c); //prints the characters one by one
        fr.close();
    }
}
```

This will produce the following result:

```
This
is
an
example
```


FileWriter Class

This class inherits from the `OutputStreamWriter` class. The class is used for writing streams of characters.

This class has several constructors to create required objects. Following is a list.

Sr. No.	Constructors and Description
1	FileWriter(File file) This constructor creates a <code>FileWriter</code> object given a <code>File</code> object.
2	FileWriter(File file, boolean append) This constructor creates a <code>FileWriter</code> object given a <code>File</code> object with a boolean indicating whether or not to append the data written.
3	FileWriter(FileDescriptor fd) This constructor creates a <code>FileWriter</code> object associated with the given file descriptor.
4	FileWriter(String fileName) This constructor creates a <code>FileWriter</code> object, given a file name.
5	FileWriter(String fileName, boolean append) This constructor creates a <code>FileWriter</code> object given a file name with a boolean indicating whether or not to append the data written.

Once you have *FileWriter* object in hand, then there is a list of helper methods, which can be used to manipulate the files.

Sr. No.	Methods with Description
1	public void write(int c) throws IOException Writes a single character.
2	public void write(char [] c, int offset, int len)

	Writes a portion of an array of characters starting from offset and with a length of len.
3	public void write(String s, int offset, int len) Write a portion of a String starting from offset and with a length of len.

Example

Following is an example to demonstrate class:

```
import java.io.*;

public class FileRead{

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

        File file = new File("Hello1.txt");
        // creates the file
        file.createNewFile();
        // creates a FileWriter Object
        FileWriter writer = new FileWriter(file);
        // Writes the content to the file
        writer.write("This\n is\n an\n example\n");
        writer.flush();
        writer.close();

        //Creates a FileReader Object
        FileReader fr = new FileReader(file);
        char [] a = new char[50];
        fr.read(a); // reads the content to the array
        for(char c : a)
            System.out.print(c); //prints the characters one by one
        fr.close();
    }
}
```

This will produce the following result:

```
This
is
an
example
```

Directories in Java

A directory is a **File** which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on **File** object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories:

- The **mkdir()** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the **File** object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **makedirs()** method creates both a directory and all the parents of the directory.

Following example creates `"/tmp/user/java/bin"` directory:

```
import java.io.File;

public class CreateDir {
    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);
        // Create directory now.
        d.mkdirs();
    }
}
```

Compile and execute the above code to create `"/tmp/user/java/bin"`.

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list()** method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;
public class ReadDir {
    public static void main(String[] args) {

        File file = null;
        String[] paths;
        try{
            // create new file object
            file = new File("/tmp");

            // array of files and directory
            paths = file.list();

            // for each name in the path array
            for(String path:paths)
            {
                // prints filename and directory name
                System.out.println(path);
            }
        }catch(Exception e){
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory:

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

19. Java – Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions:** A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FileNotFoundException_Demo {

    public static void main(String args[]){
        File file=new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }

}
```

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

```
C:\>javac FileNotFound_Demo.java
FileNotFound_Demo.java:8: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
    FileReader fr = new FileReader(file);
                        ^
1 error
```

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

- **Unchecked exceptions:** An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *`ArrayIndexOutOfBoundsException`* occurs.

```
public class Unchecked_Demo {

    public static void main(String args[]){
        int num[]={1,2,3,4};
        System.out.println(num[5]);
    }

}
```

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

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

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

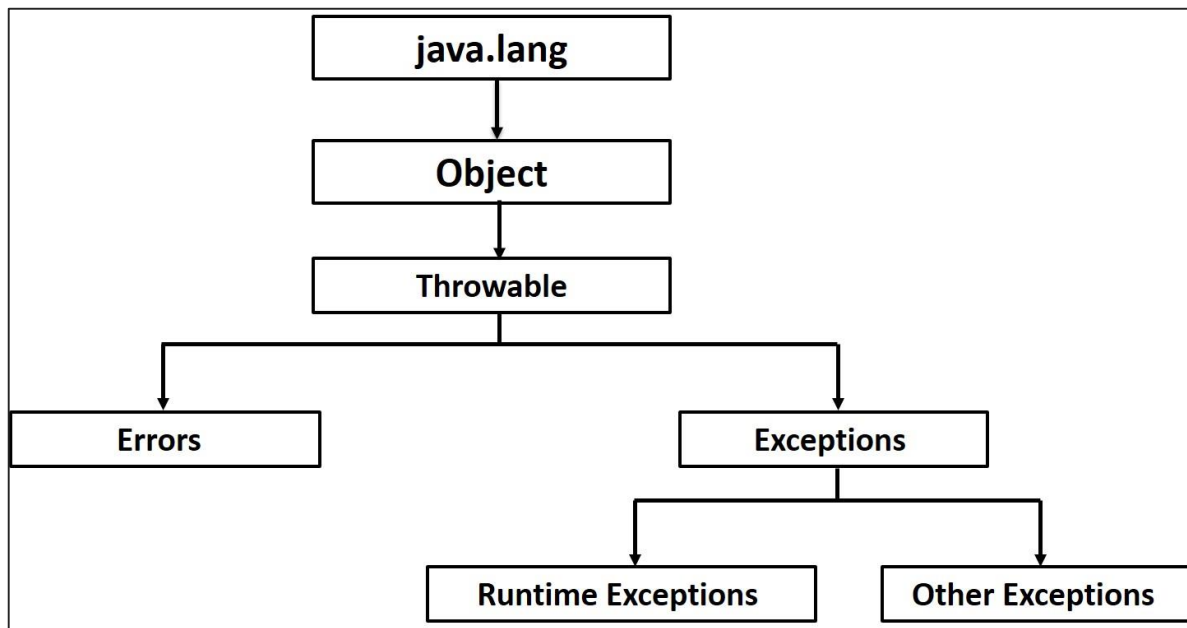
Exception Hierarchy

All exception classes are subtypes of the `java.lang.Exception` class. The exception class is a subclass of the `Throwable` class. Other than the exception class there is another subclass called `Error` which is derived from the `Throwable` class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the

runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

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



Following is a list of most common checked and unchecked [Java's Built-in Exceptions](#).

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 the 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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

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

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Exceptions Methods

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

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

Catching Exceptions

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

```
try
{
    //Protected code
}catch(ExceptionName e1)
{
    //Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

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

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

    public static void main(String args[]){
        try{
            int a[] = new int[2];
            System.out.println("Access element three :" + a[3]);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown :" + e);
        }
        System.out.println("Out of the block");
    }
}
```

This will produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

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

```
try
{
    //Protected code
}catch(ExceptionType1 e1)
{
    //Catch block
}catch(ExceptionType2 e2)
{
    //Catch block
}catch(ExceptionType3 e3)
{
    //Catch block
}
```

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

Example

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

```
try
{
    file = new FileInputStream(fileName);
    x = (byte) file.read();
}catch(IOException i)
{
    i.printStackTrace();
}
```

```

        return -1;
    }catch(FileNotFoundException f) //Not valid!
    {
        f.printStackTrace();
        return -1;
    }

```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it:

```

catch (IOException|FileNotFoundException ex) {
    logger.log(ex);
    throw ex;
}

```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException:

```

import java.io.*;
public class className
{
    public void deposit(double amount) throws RemoteException
    {
        // Method implementation
        throw new RemoteException();
    }
    //Remainder of class definition
}

```

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

```
import java.io.*;
public class className
{
    public void withdraw(double amount) throws RemoteException,
                                   InsufficientFundsException
    {
        // Method implementation
    }
    //Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
    //Protected code
}catch(ExceptionType1 e1)
{
    //Catch block
}catch(ExceptionType2 e2)
{
    //Catch block
}catch(ExceptionType3 e3)
{
    //Catch block
}finally
{
    //The finally block always executes.
}
```

Example

```
public class ExcepTest{

    public static void main(String args[]){
        int a[] = new int[2];
        try{
            System.out.println("Access element three :" + a[3]);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown  :" + e);
        }
        finally{
            a[0] = 6;
            System.out.println("First element value: " +a[0]);
            System.out.println("The finally statement is executed");
        }
    }
}
```

This will produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following:

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

The try-with-resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

    public static void main(String args[]){
        FileReader fr=null;
        try{
            File file=new File("file.txt");
            fr = new FileReader(file);  char [] a = new char[50];
            fr.read(a); // reads the content to the array
            for(char c : a)
                System.out.print(c); //prints the characters one by one
        }catch(IOException e){
            e.printStackTrace();
        }
        finally{
            try{
                fr.close();
            }catch(IOException ex){
                ex.printStackTrace();
            }
        }
    }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

```
try(FileReader fr=new FileReader("file path"))
{
    //use the resource
}catch(){
    //body of catch
}
}
```

Following is the program that reads the data in a file using try-with-resources statement.

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

    public static void main(String args[]){

        try(FileReader fr=new FileReader("E://file.txt")){
            char [] a = new char[50];
            fr.read(a); // reads the content to the array
            for(char c : a)
                System.out.print(c); //prints the characters one by one
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
- You can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.

- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{  
}
```

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

Example

```
// File Name InsufficientFundsException.java  
import java.io.*;  
  
public class InsufficientFundsException extends Exception  
{  
    private double amount;  
    public InsufficientFundsException(double amount)  
    {  
        this.amount = amount;  
    }  
    public double getAmount()  
    {  
        return amount;  
    }  
}
```

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

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
    private double balance;
    private int number;

    public CheckingAccount(int number)
    {
        this.number = number;
    }

    public void deposit(double amount)
    {
        balance += amount;
    }

    public void withdraw(double amount) throws InsufficientFundsException
    {
        if(amount <= balance)
        {
            balance -= amount;
        }
        else
        {
            double needs = amount - balance;
            throw new InsufficientFundsException(needs);
        }
    }

    public double getBalance()
    {
        return balance;
    }
}
```

```

    public int getNumber()
    {
        return number;
    }
}

```

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

```

// File Name BankDemo.java
public class BankDemo
{
    public static void main(String [] args)
    {
        CheckingAccount c = new CheckingAccount(101);
        System.out.println("Depositing $500...");
        c.deposit(500.00);

        try
        {
            System.out.println("\nWithdrawing $100...");
            c.withdraw(100.00);
            System.out.println("\nWithdrawing $600...");
            c.withdraw(600.00);
        } catch (InsufficientFundsException e)
        {
            System.out.println("Sorry, but you are short $" + e.getAmount());
            e.printStackTrace();
        }
    }
}

```

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

```

Depositing $500...

Withdrawing $100...

Withdrawing $600...

```

```
Sorry, but you are short $200.0
InsufficientFundsException
    at CheckingAccount.withdraw(CheckingAccount.java:25)
    at BankDemo.main(BankDemo.java:13)
```

Common Exceptions

In Java, it is possible to define two categories of Exceptions and Errors.

- **JVM Exceptions:** These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`.
- **Programmatic Exceptions:** These exceptions are thrown explicitly by the application or the API programmers. Examples: `IllegalArgumentException`, `IllegalStateException`.

20. Java – Inner Classes

In this chapter, we will discuss inner classes of Java.

Nested Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

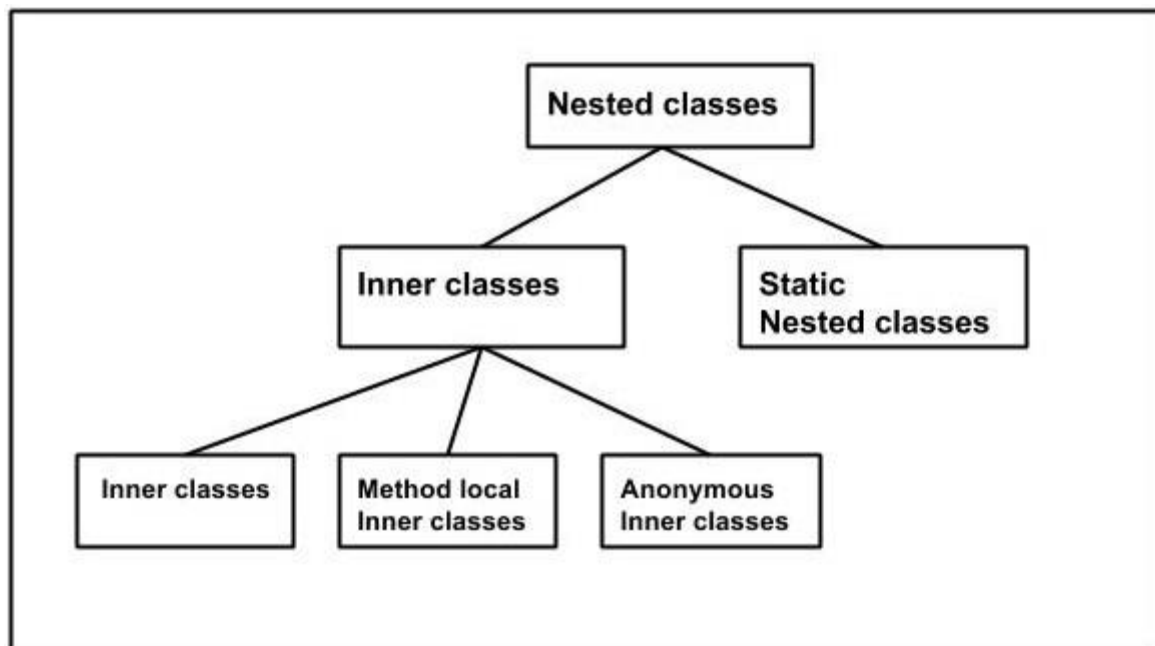
Syntax

Following is the syntax to write a nested class. Here, the class **Outer_Demo** is the outer class and the class **Inner_Demo** is the nested class.

```
class Outer_Demo{  
    class Nested_Demo{  
    }  
}
```

Nested classes are divided into two types:

- **Non-static nested classes:** These are the non-static members of a class.
- **Static nested classes:** These are the static members of a class.



Inner Classes (Non-static Nested Classes)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are:

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

```
class Outer_Demo{
    int num;
    //inner class
    private class Inner_Demo{
        public void print(){
            System.out.println("This is an inner class");
        }
    }
    //Accessing the inner class from the method within
    void display_Inner(){
        Inner_Demo inner = new Inner_Demo();
        inner.print();
    }
}

public class My_class{
    public static void main(String args[]){
        //Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();
        //Accessing the display_Inner() method.
        outer.display_Inner();
    }
}
```

```
}
}
```

Here you can observe that **Outer_Demo** is the outer class, **Inner_Demo** is the inner class, **display_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

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

```
This is an inner class.
```

Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the **getValue()** method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer=new Outer_Demo();
Outer_Demo.Inner_Demo inner=outer.new Inner_Demo();
```

The following program shows how to access the private members of a class using inner class.

```
class Outer_Demo {
    //private variable of the outer class
    private int num= 175;
    //inner class
    public class Inner_Demo{
        public int getNum(){
            System.out.println("This is the getnum method of the inner class");
            return num;
        }
    }
}

public class My_class2{
    public static void main(String args[]){
        //Instantiating the outer class
        Outer_Demo outer=new Outer_Demo();
        //Instantiating the inner class
```

```

        Outer_Demo.Inner_Demo inner=outer.new Inner_Demo();
        System.out.println(inner.getNum());
    }
}

```

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

```
The value of num in the class Test is: 175
```

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

```

public class Outerclass{

    //instance method of the outer class
    void my_Method(){
        int num = 23;

        //method-local inner class
        class MethodInner_Demo{
            public void print(){
                System.out.println("This is method inner class "+num);
            }
        }//end of inner class

        //Accessing the inner class
        MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }

    public static void main(String args[]){
        Outerclass outer = new Outerclass();
        outer.my_Method();
    }
}

```


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

This is method inner class 23

Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows:

```
AnonymousInner an_inner = new AnonymousInner(){
    public void my_method(){
        .....
        .....
    }
};
```

The following program shows how to override the method of a class using anonymous inner class.

```
abstract class AnonymousInner{
    public abstract void mymethod();
}

public class Outer_class {
    public static void main(String args[]){
        AnonymousInner inner = new AnonymousInner(){
            public void mymethod(){
                System.out.println("This is an example of anonymous inner class");
            }
        };
        inner.mymethod();
    }
}
```

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

This is an example of anonymous inner class

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument:

```
obj.my_Method(new My_Class(){
    public void Do(){
        .....
        .....
    }
});
```

The following program shows how to pass an anonymous inner class as a method argument.

```
//interface
interface Message{
    String greet();
}

public class My_class {
    //method which accepts the object of interface Message
    public void displayMessage(Message m){
        System.out.println(m.greet() + ", This is an example of anonymous inner
        calss as an argument");
    }

    public static void main(String args[]){
        //Instantiating the class
        My_class obj = new My_class();

        //Passing an anonymous inner class as an argument
        obj.displayMessage(new Message(){
            public String greet(){
                return "Hello";
            }
        });
    }
}
```

```
}
}
```

If you compile and execute the above program, it gives you the following result.

```
Hello This is an example of anonymous inner class as an argument
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows:

```
class MyOuter {
    static class Nested_Demo{
    }
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

```
public class Outer{
    static class Nested_Demo{
        public void my_method(){
            System.out.println("This is my nested class");
        }
    }
    public static void main(String args[]){
        Outer.Nested_Demo nested = new Outer.Nested_Demo();
        nested.my_method();
    }
}
```

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

```
This is my nested class
```


Java - Object Oriented

21. Java – Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

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

```
class Super{
    .....
    .....
}

class Sub extends Super{
    .....
    .....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using `extends` keyword, the `My_Calculation` inherits the methods `addition()` and `Subtraction()` of `Calculation` class.

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

```
class Calculation{
    int z;

    public void addition(int x, int y){
        z = x+y;
        System.out.println("The sum of the given numbers:"+z);
    }

    public void Substraction(int x,int y){
        z = x-y;
        System.out.println("The difference between the given numbers:"+z);
    }
}

public class My_Calculation extends Calculation{

    public void multiplication(int x, int y){
        z = x*y;
        System.out.println("The product of the given numbers:"+z);
    }

    public static void main(String args[]){
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Substraction(a, b);
        demo.multiplication(a, b);
    }
}
```

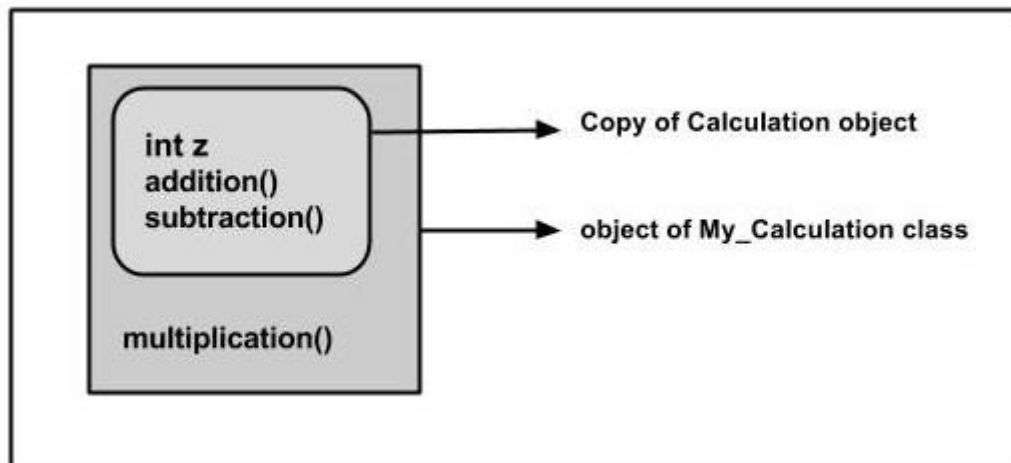
Compile and execute the above code as shown below.

```
javac My_Calculation.java
java My_Calculation
```


After executing the program, it will produce the following result.

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

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



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass **My_Calculation**.

```
Calculation cal = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);
```

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

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

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

```
super.variable  
super.method();
```

Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named `display()` with different implementations, and a variable named `num` with different values. We are invoking `display()` method of both classes and printing the value of the variable `num` of both classes. Here you can observe that we have used `super` keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name `Sub_class.java`.

```
class Super_class{  
  
    int num = 20;  
  
    //display method of superclass  
    public void display(){  
        System.out.println("This is the display method of superclass");  
    }  
}  
  
public class Sub_class extends Super_class {  
  
    int num = 10;  
  
    //display method of sub class  
    public void display(){  
        System.out.println("This is the display method of subclass");  
    }  
}
```

```

public void my_method(){

    //Instantiating subclass
    Sub_class sub = new Sub_class();

    //Invoking the display() method of sub class
    sub.display();

    //Invoking the display() method of superclass
    super.display();

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

    //printing the value of variable num of superclass
    System.out.println("value of the variable named num in super class:"+
super.num);
}

public static void main(String args[]){
    Sub_class obj = new Sub_class();
    obj.my_method();

}
}

```

Compile and execute the above code using the following syntax.

```

javac Super_Demo
java Super

```

On executing the program, you will get the following result –

```

This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20

```

Invoking Superclass Constructor

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

```
super(values);
```

Sample Code

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

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

```
class Superclass{

    int age;

    Superclass(int age){
        this.age = age;
    }

    public void getAge(){
        System.out.println("The value of the variable named age in super class
is: " +age);
    }

}

public class Subclass extends Superclass {

    Subclass(int age){
        super(age);
    }

}
```

```

    public static void main(String argd[]){
        Subclass s = new Subclass(24);
        s.getAge();
    }
}

```

Compile and execute the above code using the following syntax.

```

javac Subclass
java Subclass

```

On executing the program, you will get the following result –

```

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

```

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
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal{
}

class Mammal extends Animal{
}

class Reptile extends Animal{
}

public class Dog extends Mammal{

    public static void main(String args[]){

        Animal a = new 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 will produce the following result –

```
true
true
true
```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {  
}  
  
public class Mammal implements Animal{  
}  
  
public class Dog extends Mammal{  
}
```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

```
interface Animal{  
  
}  
  
class Mammal implements Animal{  
  
}  
  
public class Dog extends Mammal{  
  
}
```

```
public static void main(String args[]){  
  
    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 will 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 look into an example –

```
public class Vehicle{}
public class Speed{}

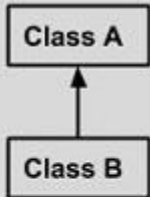
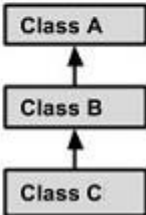
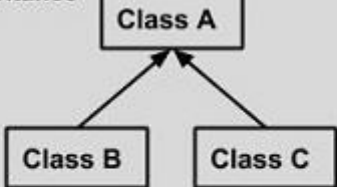
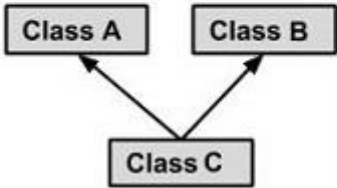
public class Van extends Vehicle{
    private Speed sp;
}
```

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.

Types of Inheritance

There are various types of inheritance as demonstrated below.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {..... } </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {..... } </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support multiple Inheritance </pre>

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal –

```
public class extends Animal, Mammal{}
```

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

22. Java – Overriding

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

Example

Let us look at an example.

```
class Animal{

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

class Dog extends Animal{

    public void move(){
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog{

    public static void main(String args[]){
        Animal a = new Animal(); // Animal reference and object
        Animal b = new Dog(); // Animal reference but Dog object

        a.move();// runs the method in Animal class
        b.move();//Runs the method in Dog class
    }
}
```

This will produce the following result:

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

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

Therefore, in the above example, the program will compile properly since `Animal` class has the method `move`. Then, at the runtime, it runs the method specific for that object.

Consider the following example:

```
class Animal{

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

class Dog extends Animal{

    public void move(){
        System.out.println("Dogs can walk and run");
    }
    public void bark(){
        System.out.println("Dogs can bark");
    }
}

public class TestDog{

    public static void main(String args[]){
        Animal a = new Animal(); // Animal reference and object
        Animal b = new Dog(); // Animal reference but Dog object

        a.move();// runs the method in Animal class
        b.move();//Runs the method in Dog class
    }
}
```

```
        b.bark();  
    }  
}
```

This will produce the following result:

```
TestDog.java:30: cannot find symbol  
symbol   : method bark()  
location: class Animal  
        b.bark();  
        ^
```

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

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the 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 unchecked 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 will produce the following result:

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

23. Java – Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}  
public class Animal{}  
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
Deer d = new Deer();  
Animal a = d;  
Vegetarian v = d;  
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

```
/* File name : Employee.java */
public class Employee
{
    private String name;
    private String address;
    private int number;
    public Employee(String name, String address, int number)
    {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }
    public void mailCheck()
    {
        System.out.println("Mailing a check to " + this.name
            + " " + this.address);
    }
    public String toString()
    {
        return name + " " + address + " " + number;
    }
    public String getName()
    {
        return name;
    }
    public String getAddress()
    {
        return address;
    }
}
```

```

    public void setAddress(String newAddress)
    {
        address = newAddress;
    }
    public int getNumber()
    {
        return number;
    }
}

```

Now suppose we extend Employee class as follows:

```

/* File name : Salary.java */
public class Salary extends Employee
{
    private double salary; //Annual salary
    public Salary(String name, String address, int number, double
        salary)
    {
        super(name, address, number);
        setSalary(salary);
    }
    public void mailCheck()
    {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }
    public double getSalary()
    {
        return salary;
    }

    public void setSalary(double newSalary)
    {
        if(newSalary >= 0.0)
        {
            salary = newSalary;

```



```

    }
}
public double computePay()
{
    System.out.println("Computing salary pay for " + getName());
    return salary/52;
}
}

```

Now, you study the following program carefully and try to determine its output:

```

/* File name : VirtualDemo.java */
public class VirtualDemo
{
    public static void main(String [] args)
    {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

This will produce the following result:

```

Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
ailing check to Mohd Mohtashim with salary 3600.0
Call mailCheck using Employee reference--
Within mailCheck of Salary class
ailing check to John Adams with salary 2400.0

```

Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheck()*, the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time. Invoking

mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

24. Java – Abstraction

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
/* File name : Employee.java */
public abstract class Employee
{
    private String name;
    private String address;
    private int number;
    public Employee(String name, String address, int number)
```

```
{
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
}
public double computePay()
{
    System.out.println("Inside Employee computePay");
    return 0.0;
}
public void mailCheck()
{
    System.out.println("Mailing a check to " + this.name
        + " " + this.address);
}
public String toString()
{
    return name + " " + address + " " + number;
}
public String getName()
{
    return name;
}
public String getAddress()
{
    return address;
}
public void setAddress(String newAddress)
{
    address = newAddress;
}
```

```

    public int getNumber()
    {
        return number;
    }
}

```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way:

```

/* File name : AbstractDemo.java */
public class AbstractDemo
{
    public static void main(String [] args)
    {
        /* Following is not allowed and would raise error */
        Employee e = new Employee("George W.", "Houston, TX", 43);

        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

When you compile the above class, it gives you the following error:

```

Employee.java:46: Employee is abstract; cannot be instantiated
        Employee e = new Employee("George W.", "Houston, TX", 43);
                        ^
1 error

```

Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way:

```
/* File name : Salary.java */
public class Salary extends Employee
{
    private double salary; //Annual salary
    public Salary(String name, String address, int number, double
        salary)
    {
        super(name, address, number);
        setSalary(salary);
    }
    public void mailCheck()
    {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }
    public double getSalary()
    {
        return salary;
    }
    public void setSalary(double newSalary)
    {
        if(newSalary >= 0.0)
        {
            salary = newSalary;
        }
    }
    public double computePay()
    {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
/* File name : AbstractDemo.java */
public class AbstractDemo
{
    public static void main(String [] args)
    {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();

        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}
```

This produces the following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
ailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
ailing check to John Adams with salary 2400.
```

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.

- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.

Following is an example of the abstract method.

```
public abstract class Employee
{
    private String name;
    private String address;
    private int number;

    public abstract double computePay();

    //Remainder of class definition
}
```

Declaring a method as abstract has two consequences:

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note: Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below:

```
/* File name : Salary.java */
public class Salary extends Employee
{
    private double salary; // Annual salary

    public double computePay()
    {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }

    //Remainder of class definition
}
```


25. Java – Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

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

To achieve encapsulation in Java:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example

Following is an example that demonstrates how to achieve Encapsulation in Java:

```
/* File name : EncapTest.java */
public class EncapTest{

    private String name;
    private String idNum;
    private int age;

    public int getAge(){
        return age;
    }

    public String getName(){
        return name;
    }

    public String getIdNum(){
        return idNum;
    }
}
```

```

    public void setAge( int newAge){
        age = newAge;
    }

    public void setName(String newName){
        name = newName;
    }

    public void setIdNum( String newId){
        idNum = newId;
    }
}

```

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program:

```

/* File name : RunEncap.java */
public class RunEncap{

    public static void main(String args[]){
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " +
        encap.getAge());
    }
}

```

This will produce the following result:

```
Name : James Age : 20
```

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

26. Java – Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

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

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

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

Declaring Interfaces

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

Example

Following is an example of an interface:

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
    //Any number of final, static fields
    //Any number of abstract method declarations\
}
```

Interfaces have the following properties:

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

Example

```
/* File name : Animal.java */
interface Animal {

    public void eat();
    public void travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

This will produce the following result:

```
Mammal eats
Mammal travels
```

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

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementing interfaces, there are several rules:

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

Extending Interfaces

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

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

```
//Filename: Sports.java
public interface Sports
{
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports
{
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports
{
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
}
```

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

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The `extends` keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the `MouseListener` interface in the `java.awt.event` package extended `java.util.EventListener`, which is defined as:

```
package java.util;  
  
public interface EventListener  
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

Creates a common parent: As with the `EventListener` interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends `EventListener`, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class: This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

27. Java – Packages

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

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

Some of the existing packages in Java are:

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

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

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

Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

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

Example

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

Following package example contains interface named *animals*:

```
/* File name : Animal.java */
package animals;
interface Animal {
    public void eat();
    public void travel();
}
```

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

```
package animals;

/* File name : MammalInt.java */
public class MammalInt implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

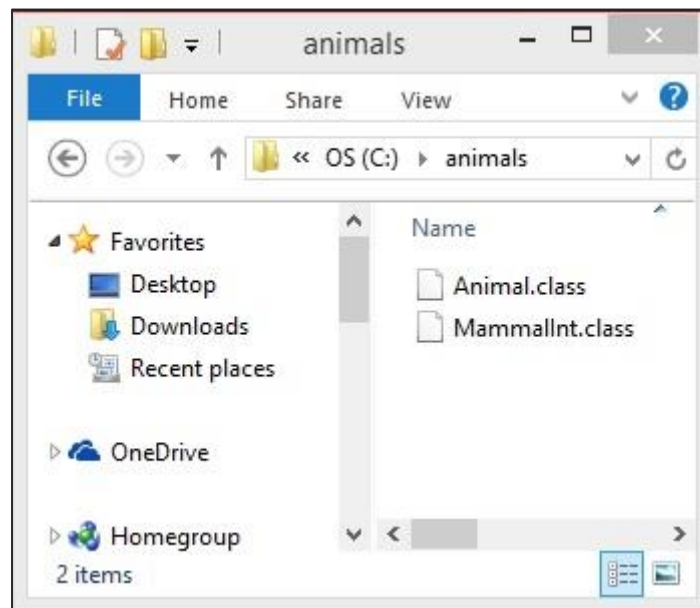
    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

Now compile the java files as shown below:

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

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



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

```
$ java animals.MammalInt  
ammal eats  
ammal travels
```

The import Keyword

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

Example

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

```
package payroll;  
  
public class Boss  
{  
    public void payEmployee(Employee e)  
    {  
        e.mailCheck();  
    }  
}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

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

```
payroll.Employee
```

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

```
import payroll.*;
```

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

```
import payroll.Employee;
```

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

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 follows:

- 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
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**.

For example:

```
// File Name: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

Now, compile this file as follows using -d option:

```
$javac -d . Dell.java
```

The files will be compiled as follows:

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in `\com\apple\computers\` as follows:

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

```
<path-one>\sources\com\apple\computers\Dell.java  
  
<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give access to the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):

- In Windows -> C:\> set CLASSPATH
- In UNIX -> % echo \$CLASSPATH

To delete the current contents of the CLASSPATH variable, use:

- In Windows -> C:\> set CLASSPATH=
- In UNIX -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

- In Windows -> set CLASSPATH=C:\users\jack\java\classes
- In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH