BASICS OF JAVA

ARRAYS - String- StringBuffer

OBJECT AND CLASSES

INHERITANCE

INTERFACE

EXCEPTION HANDLING

1. A Brief History of Java

Java was developed at Sun Microsystems in 1991, by a team comprising James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan as its members. The language was initially called Oak. It was later termed as Java. Java was launched on 23 May, 1995. The Java software was released as a development kit. The first two versions were named JDK 1.0 and JDK 1.1. In 1998, while releasing the next version, Sun Microsystems changed the nomenclature from Java Development Kit (JDK) to Software Development Kit (SDK). Also, it added "2" to the name. The released version of Java was called Java 2 SDK 1.2.

With every version, Java became stronger and stronger. At the time of writing of the book, Java had come up with its version 6, known as JDK 6. Java has already released the beta of its next version. In our text, we will be describing version 6.

The following table illustrates the various versions released till date:

Version	Release date	Major additions
JDK 1.0	January 1996	Initial release
JDK 1.1	February 1997	Inner classes, JavaBeans, JDBC, RMI
J2SE 1.2	December 1998	Swing and Collections Framework
J2SE 1.3	May 2000	HotSpot JVM, RMI, JavaSound
J2SE 1.4	February 2002	Regular expressions, Java Web Start
J2SE 5.0	September 2004	Generics, autoboxing, enumeration
Java SE 6	December 2006	Database manager and many new facilities
Java SE 7		Current

Why Is Java Popular for the Internet?

The main reason why Java is popular on the Internet is that Java offers applets. Applets are tiny programs which run inside the browser. Before the introduction of applets, the web pages were mostly static. After the use of this application, they have become dynamic, making browsing richer.

Applets are very safe. They cannot write to the local hard disk. Hence, there is no threat of viruses while surfing the Internet. This has made applets (and thereby Java) highly popular on the Internet.

However, this is only one aspect. Because applets are Java programs, they inherit strengths of Java. They also support graphical user interface (GUI). Hence, surfing the Internet can be quite enjoying when web pages use applets. Please note that applets are only one aspect of Java.

Java is a complete programming language. We can write applications in different walks of life. As we move along the text, we will be discussing them thoroughly.

Features of Java

The features of Java, which make Java a powerful, popular language, can be stated briefly as

- Simple
- Portable
- Robust
- Architecture neutral
- Distributed
- Secure
- Object oriented
- Multi-threaded
- Interpreted

Simple

Java is simpler because of many reasons. The most important thing is the absence of pointers. Many unnecessary features of C/C++, like overloading of operators, are removed from Java.

Secure

Java language becomes secure because of the following properties/components:

- No pointers
- Bytecode verifier
- Class loader
- Security manager

Portable

The Java code is highly portable. Write Once Run Anywhere (WORA) is the basic Java slogan. Every platform will always have its own Java runtime machine. What you need is the Java bytecode. You are free to write the Java source code on any platform and compile to bytecode there. The bytecode so generated is platform independent. It will run on any platform.

Object oriented

Java is object oriented. During the last many years, every new language introduced is object oriented.

Robust

Java is a robust language. It does not crash the computer, even when it encounters minor mistakes in a program. It has the ability to withstand the threats. This is considered as a great advantage for a programming language.

Multi-threaded

With the introduction of high-speed microprocessors a few years ago, users wanted to perform many tasks at a time. This is possible on a single-chip processor only by the use of multi-threading. Multi-threading means ability of a program to run multiple (more than one) pieces of program code simultaneously. This is possible by time slicing in a single processor system.

Every thread is assigned a small time slice to run. It creates a feeling that all the tasks are running simultaneously in time

.

Interpreted

Java is an interpreted language. When we write a program, it is compiled into a class file. The interpreter executes this class file. However, the interpreters 30 years ago were interpreting the statement in textual form. It was a very slow process. Java interprets bytecode; hence, it is considerably fast. Actually, Java gets all the advantages of interpretation, without suffering from major disadvantages.

Architecture neutral

The Java programming language is dependent neither on any particular microprocessor family, nor on any particular architecture. Any standard computer or even a microcomputer can run Java programs. The mobile phones also have software based on Java.

Distributed

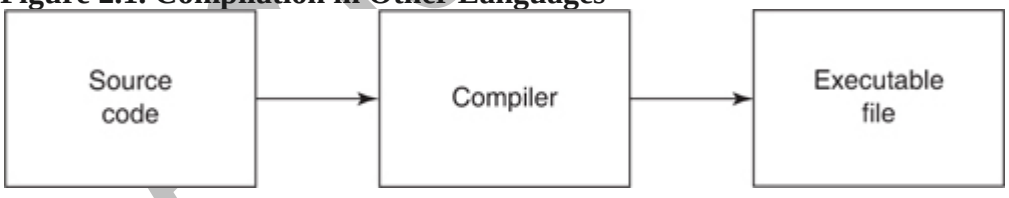
Java has many in-built faculties which make it easy to use language in distributed systems.

How Java Programs Run

Earlier we had said that Java is an interpreted language. Some of us may be familiar with languages C and C++, which are not interpreted languages. Let us try to understand the difference.

While developing a program in non-interpreted languages (like C/C++), the following steps occur. A program is written in higher level language (HLL). It is called the source code, typically a `.c` or `.cpp` file. Next, this program is compiled completely, resulting in an executable file, as shown in Figure 2.1. It typically has an `.exe` extension. This file contains a machine language program (sometimes called machine code). This file can be executed on machine (computer) with the help of an operating system.

Figure 2.1. Compilation in Other Languages



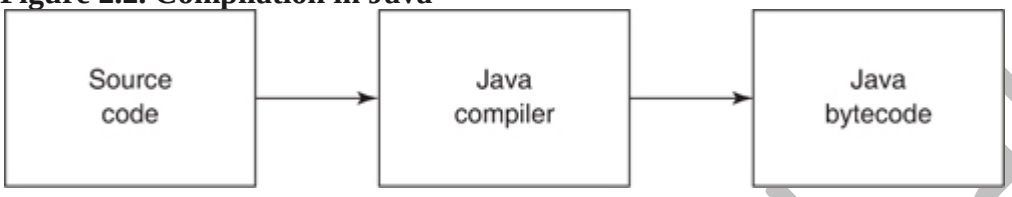
To understand how Java programs run, we must understand two new terms. These are *bytecode* and *virtual machine* (VM).

- Bytecode: This code is actually platform independent. We can consider it as a machine code for a hypothetical computer. When you compile an `anyName.java` file successfully, you get this code. It is in the form of `anyName.class` file.
- Virtual machine: The name virtual suggests that this is not an actual machine. It is just a program which simulates a machine on which Java programs are supposed to run.

Actually, you can suppose that VM is a hardware which runs programs in bytecode (as its own machine language).

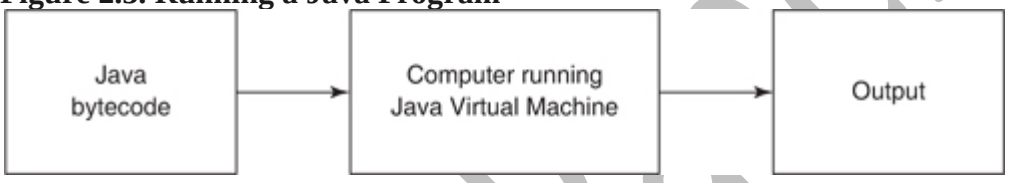
Now we can understand how Java programs are executed. First, a program is written in Java. It is called the source code. The file has `.java` extension. Next, this program is compiled by a Java compiler. It produces a bytecode. The file extension is `.class`. This process is shown in Figure 2.2.

Figure 2.2. Compilation in Java



The computer does not directly execute this bytecode. Instead, a program known as Java Virtual Machine (JVM) runs on a computer. We pass this bytecode as an input to this program. This program interprets and executes this bytecode. This fact is explained in Figure 2.3.

Figure 2.3. Running a Java Program



Advantages and Disadvantages of an Interpreted Language

In the early days of computing, the compilation of interpreted languages was quite different. The source code was executed line by line by a program called interpreter. As the interpreter had to interpret (understand) and then execute every line, the speed of execution was very poor. Although Java is an interpreted language, it uses a slightly different mechanism. A program is written as a source code, typically a `.java` file. When it is compiled, it results in another file that has `.class` extension. This file contains a code known as the bytecode. It is a code for a hypothetical machine. This file is executed by another program called interpreter. Since the interpreter interprets the bytecode (and not the text lines), the speed of execution is much higher as compared to the old interpreted languages.

Still today, Java programs run at lesser speed as compared to other language programs which are completely compiled.

Third-party softwares have already started appearing which compile and produce executable code from bytecode. Because of this and other developments that will occur in future, Java programs are expected to run as fast as any other language programs.

However, the major advantage of Java being an interpreted language is that it is machine (hardware platform) independent. If there is any computer in the world of any specific type (make/hardware) and if a JVM exists for that machine, then the Java bytecode runs on that machine in exactly the same way as being run on any other machine. In today's world of the Internet, where heterogeneous collection of hardware and software platforms exists, this machine independence is considered as a great advantage.

Advantages of Java Technology

- Get started quickly: Although the Java programming language is a powerful object-oriented language, it's easy to learn, especially for programmers already familiar with C or C++.
- Write less code: Comparisons of program metrics (class counts, method counts, and so on) suggest that a program written in the Java programming language can be four times smaller than the same program in C++.
- Write better code: The Java programming language encourages good coding practices, and its garbage collection helps you avoid memory leaks. Its object orientation, its Java Beans component architecture, and its wide-ranging, easily extendible API let you reuse other people's tested code and introduce fewer bugs.
- Develop programs faster: Your development time may be as much as twice as fast versus writing the same program in C++. Why? You write fewer lines of code and it is a simpler programming language than C++.
- Avoid platform dependencies with 100% Pure Java: You can keep your program portable by following the purity tips mentioned throughout this tutorial and avoiding the use of libraries written in other languages.
- Write once, run anywhere: Because 100% Pure Java programs are compiled into machine-independent bytecodes, they run consistently on any Java platform.
- Distribute software more easily: You can upgrade applets easily from a central server. Applets take advantage of the feature of allowing new classes to be loaded "on the fly," without recompiling the entire program.

Java Platform

The Java platform consists of the following.

- A language for writing programs
- A set of APIs
- Class libraries
- Other programs used in developing, compiling, and error-checking
- A VM which loads and executes the class files

The Java language is the one which we are going to study in detail in this book. Language consists of syntax (rules) and semantics. Syntax means rules of grammar. Semantics means the description of what will happen when statements (instructions) that follow syntax rules are executed.

API as defined by Java is "the specification of how a programmer writing an application accesses the behaviour and state of classes and objects". This is rather a terse definition. While writing Java programs we need many classes which are designed and developed by Java language creators. Programs simply would not run without them. The library of such classes is nothing but API.

To develop Java programs, we need a compiler, and other programs and tools. They are also part and parcel of the Java platform.

A program named JVM is needed to execute the Java programs. This is the most important component of the platform.

Java Development Kit

The JDK is a development environment for building applications, applets, and components using the Java programming language. In more technical terms, the current version is called Java™ Platform, Standard Edition 6 JDK. For simplicity, let us refer it as JDK 6. It is possible that you have heard the term SDK. You might wonder what is the difference between JDK and SDK. Actually, they mean the same thing. The JDK 6 includes tools useful for developing and testing programs written in Java. They help us running them on the Java platform. These tools are designed to be used from the command line. They do not provide a GUI. The only exception is the "applet viewer". Let us describe the contents of JDK 6 in detail.

1. Development tools: Tools and utilities that will help one to develop, execute, debug, and document programs written in the Java programming language are provided in the "bin" sub-directory. These tools are the foundation of the JDK 6. The basic tools are described in the following table.

Tool	Description
javac	The compiler for the Java programming language
java	The launcher for Java applications. In this release, a single launcher is used for both development and deployment
javadoc	API documentation generator
appletviewer	Run and debug applets without a web browser
jar	Manage Java Archive (JAR) files
jdb	The Java Debugger
javah	C header and stub generator. Used to write native methods
javap	Class file disassembler
extcheck	Utility to detect JAR conflicts

2. Runtime environment: An implementation of the Java 2 runtime environment for use by the SDK. The runtime environment includes a JVM, class libraries, and other files that support the execution of programs written in the Java programming language. These are present in the "jre" sub-directory.
3. Additional libraries: Additional class libraries and support files required by the development tools are present in the "lib" sub-directory.
4. Demo applets and applications: Examples, with source code, (of programming for the Java platform) are present in the "demo" sub-directory. These include examples that use Swing and other Java Foundation Classes, and the Java Platform Debugger Architecture.
5. JDK documentation: The documentation is available online. It contains API specifications, feature description, developer guides, reference pages for SDK tools and utilities, demos, and links to related information. We can download this documentation and install on our machine.

Java Virtual Machine (JVM)

> *The Java Virtual Machine (JVM) is a platform-independent engine used to run Java applets and applications. The JVM knows nothing of the Java programming language, but it does understand the particular file format of the platform and implementation independent class file produced by a Java compiler. Therefore, class files produced by a Java compiler on one system can execute without change on any system that can invoke a Java Virtual Machine.*[1]
>
> *When invoked with a particular class file, the JVM loads the file, goes through a verification process to ensure system security, and executes the instructions in that class file.*

The JVM, in other words, forms a layer between the operating system and the Java program that is trying to execute. That explains how *one* Java program can run without change on a variety of systems: it can not! A Java program runs on only *one* system, namely the Java Virtual Machine. That virtual system, in turn, runs on a variety of operating systems and is programmed quite differently for various systems. To the Java programmer, it provides a unified interface to the actual system calls of the operating system.[2] You can include graphics, graphical user interface elements, multimedia, and networking operations in a Java program and the JVM will negotiate the necessary details between the class file(s) and the underlying operating system. The JVM produces exactly the same results – in theory – regardless of the underlying operating system. In the Basic (or C, or C++) programming language, for example, you can create code that specifies to multiply two integers 1000 and 2000 and store the result as another integer. That code works fine on some systems, but can produce negative numbers on others.[3] In Java, this can not happen: either the code fails on all platforms, or it works on all platforms.

Because the JVM is in effect its own computer, it can shield the actual computer it is running on from potentially harmful effects of a Java program. This is especially important because Java programs known as *applets* can *automatically* start executing on your machine when you are surfing the web if the appropriate feature of your web browser is enabled. If these programs were allowed to meddle with your system, you could accidentally execute a program that would proceed to erase your entire disk. That, of course, would prompt people to disable Java on their web browser, which in turn would be bad news for anyone who supports the Java concept.

[1] In general, the Java Virtual Machine is an abstractly specified class file interpreter that can be realized by different software makers. The JVM that comes with the JDK was created by SUN Microsystems, but any other JVM is also able to run the same class files. Different JVM's can vary in efficiency, but they all must run the same class files.

[2] This is somewhat similar to old Basic programs: a simple Basic program can run on virtually any system that has a Basic interpreter installed since the interpreter mediates between the program trying to run and the operating system.

[3] Programming languages have a *largest possible* integer whose value can differ on different systems. A C++ program executing on a machine with a largest integer bigger than 2,000,000 produces the correct result, but on a system where the largest integer is, say, 32,767 it fails. The JVM has the *same* largest integer on every platform.

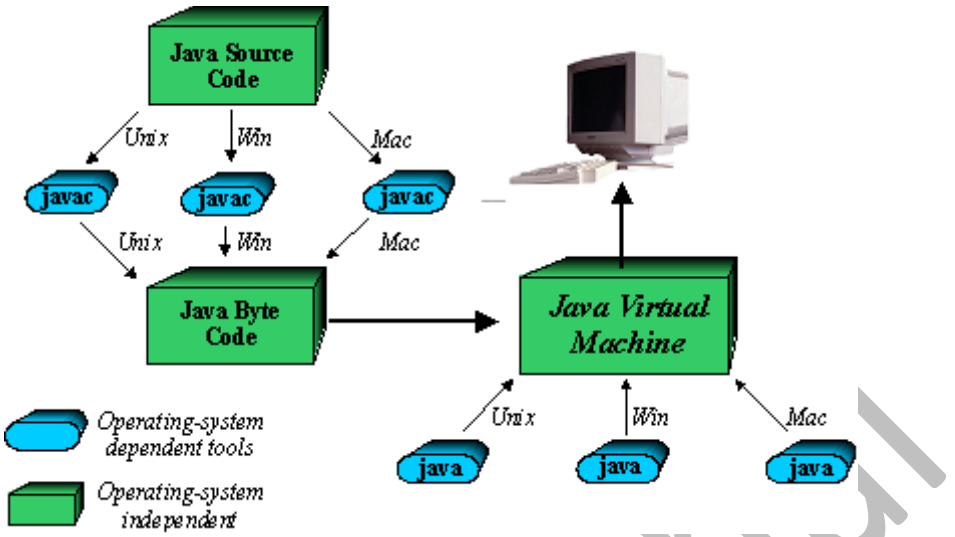


Figure 1.09: Illustrating the machine dependent/independent parts of Java programs

2. Java Language Basics

In our first Java program, we will find the area of a circle with radius 5.

Problem: Write a program to find area of a circle with radius 5.

```
//    first.java
class first
 {
public static void main( String args[] )
   {
    int r  ;
    r = 5 ;
    System.out.println(3.14 * r * r);
   }
}
```

If you run the program, you will get the following output.

Output

```
78.5
```

The output was as expected. Though the program was very simple still it had many components (parts). Let us describe its every part (Table 3.1).

Table 2.1. Program Parts Illustrated	
Program code	**Explanation**
`//  first.java`	This is a comment
`class first`	The name of the class is `first`. This program must be in a file `first.java`
`{`	Left brace: start of class
`public static void main`	The execution of program starts with method `main ()`
`( String args[] )`	
`  {`	Left brace: start of main method
`  int r ;`	Declaration of variable `r`
`  r = 5 ;`	Initialization of `r`
`System.out.println(3.14 * r * r);`	Program statement. Prints the value
`  }`	Right brace: end of method main
`}`	Right brace: end of class `first`

Java Program Structure

A useful Java program will consist of many parts. It will have many features. It will not be easy to absorb every thing at one go. Therefore, we should first think of very simple programs, then simple programs, and so on. We should study Java step by step. Now in light of this program, let us study structure of a very simple Java program.

Structure of a very simple Java program can be summarized as follows:

1. A program say `first` is written in a file named `first.java`.
2. The file contains a class `first`, as a rule.
3. The class contains method `main`. It has to be declared with parameters: `String args[]`, as shown in Table 3.1.
4. The method main contains declarations and program statements.
5. Java applications begin execution at method `main()`.

As a matter of the rule, name of the class which contains method `main()` and the name of the file (without extension) must be identical.

Please note that this is description of a very simple program. As we start studying more and more features, we will refine this definition.

Basic Building Blocks

Any Java program (or a program in any other language for that matter) consists of few building blocks like comments, declarations, statements, and methods. These blocks are formed by some small atomic quantities known as tokens. Here we will not go into the grammar in depth. Still we will discuss step by step how programs, that is declarations and statements, are written.

1. Comments

Comments are that part of the program, which are supposed to give more information (or explanation) to the reader. They increase the readability of the program. The compiler does not convert it into code. (We may say compiler ignores it.) Therefore, presence of comments has no relevance to program (execution) speed. In Java, comments are enclosed in pair of `/*` and `*/`. Such comments can be of one or more lines. It may be noted that, such comment can appear anywhere, where a white space can appear in a program.

In addition Java uses double slash (`//`) for single-line comments. Any text from `//` to the end of that line is treated as comment.

In this text, you will see every program starts with the program name as a comment. It is not merely a style. It is a discipline which helps in long run.

2. Character set

When we write a program, we actually write characters. The console input and output also uses characters. Character set stands for the collection of characters used by any programming language. Java supports Unicode character set. Unicode characters need 16 bits (instead of 8 for ASCII) to store a character. It will be heartening to know that in Unicode there is representation for scripts (languages), like Japanese, Chinese, and many Indian ones, like Devnagri and Tamil.

Earlier languages used ASCII character set. Java also supports it as Unicode encompasses the ASCII character set.

A Java program can generate all of these characters. If we talk of writing Java program, that is source code, then there are some restrictions. We cannot use each and every character in writing a program. Table 3.2 illustrates the characters used in writing a program.

Table 2.2. Characters for Writing Java Programs

Group	Details
Letters—English alphabets	a to z and A to Z
Letters—non-English alphabets	All characters that are considered to be a letter by the Unicode 2.0 standard.
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special characters I	dollar mark ($) and underscore (_) character

Table 2.2. Characters for Writing Java Programs	
Group	**Details**
Special characters II	`! @ # % ^ & * - = + \ < > / ?`
Special characters III	`{ } [ ] ( ) ; ,`
Separator	blank, tab, new line

All Unicode characters can be written to file. However, if we are writing them to screen, some of them may not be shown properly.

3.Constants

The most basic things in a program are the variables and the values they take. We have seen a statement:

```
r = 5 ;
```

Here, r is a variable and it takes value "5".

We will call this value appearing in a program as constant. (In strict computer jargon, it is called literal).

Constants can be divided into following three categories:

1. Integer constants
2. Real constants
3. Character constants

Integer constants

Integers can be expressed in three possible notations, that is octal, decimal, and hexadecimal. In daily use, we use integer constants in decimal form. These constants are either positive or negative. Plus sign is optional.

Following are examples of valid integer constants in decimal notation:

23

−55

+2367

If we have to state an integer constant in octal notation (format), we must precede it with zero. This is the way we tell the computer that we mean octal representation. Please note that digits 8 and 9 are not valid in octal notation.

Following are examples of valid integer constants in octal notation:

023

055

02367

Hexadecimal number system consist of digits 0 to 9 and alphabets "a" to "f". These constants are preceded by `0x`

Following are examples of valid integer constants in hexadecimal notation:

```
0x23
0x5B
0Xff23
```

Please note that alphabets “a” to “f” and “x” can be in either small or capital.

Real constants

Real or floating-point constants are written in either fixed point or scientific notation. Following are examples of valid real constants in fixed point notation.

2.3
−5.54
236.7

Notice the presence of decimal point. Also note, optional sign. Following are examples of valid real constants in scientific notation:

0.2e3
−5E5
2E67

Following conventions are used with scientific notation:

1. For both mantissa and exponent, sign is optional.
2. Use of capital “E” is allowed in place of small “e”.

When we write numerical constants, they may belong to different types like `float`, `double`, `integer`, or `long`. There may be some confusion in the mind of compiler. If we write 2 it can very well represent an `integer` or `long` or `float` value. Hence, Java allows us to specify the data type of a constant using a suffix. If we use suffixes f, F, l, L, d, D they tell the compiler the necessary data type of the constant. Type `float` is specified by f or F. Type `long` is specified by l or L. Similarly, type `double` is specified by d or D.

Can you figure out why type `integer` is not given a suffix? The answer is simple; the type `integer` is the default numeric type. Any constant if not specified is treated as `integer`.

Character constants

Enclosing a single character in single quotes forms simple character constants:

Example: ‘A’, ‘B’, ‘)’

There are many characters in ASCII character set which are used for special purpose (for example, carriage return, form feed). Some of them cannot be typed with a keyboard. Naturally, they cannot be placed within single quotes. Therefore, alternate notation (format) has been developed. It uses a backslash character. It is also referred as escape codes.

Table 3.3 illustrates commonly used special characters.

Table 2.3. Escape Characters	
Character	**Represented by**
Backspace	'\b'
Form feed	'\f'

Table 2.3. Escape Characters	
Character	**Represented by**
New line	'\n'
Carriage return	'\r'
Horizontal tab	'\t'
Single quote	'\''
Double quote	'\"'
Backslash	'\\'

It is interesting to note that to denote a single backslash character we use two such characters. Please note that unlike C++, Java does not define escape codes for vertical tab and the bell characters. See escape codes in action at the end of chapter example.
If we have to specify Unicode character, we have to write its hexadecimal value preceded by character u. For example,

```
'\u0100'
```

Please note that you cannot use uppercase U. Also, there must be four (no more no less) hexadecimal digits. Hence, constants like

```
'\U0100' or \u100
```

are invalid.

String constants
A collection of characters is known as `string`. Putting characters in the pair of double quotes forms a `string` constant.
Following are examples of valid `string` constants:

```
"Hello"
"World"
"good morning" *
```

Java has an in-built class `String` to deal with character `strings`. We will deal with it later in a separate chapter.

3. Variables

We have seen constants. To store their values we need what is called variables. As name suggests, computer allows us to vary (change, manipulate) the values stored in a variable. In earlier programs, we have used variable names like area1. Grammatically variable name is an identifier. Hence, rule for forming variables are same as those for forming identifiers.
Rules for forming variable names can be stated as follows:

1. Variable names are formed with the help of letters and digits.
2. Alphabets can be in upper or lower case.

3. Only letters are allowed as starting character. Underscore character (_) and Dollar symbol ($) are treated as letter as far as variables are concerned.
4. The first character cannot be a digit.
5. There is no restriction on number of characters in a variable.
6. As white space characters (such as space and tab) are used to separate tokens, they cannot be part of a variable.

Following are valid variable names:
marks
Marks
K3G
Kabhi_Kushi_Kabhi_Gum
$JamesBond
Following are invalid variable names:
mark's
4you
m.x

4. Keywords

Every language reserve few words for its own use. They are termed as keywords or reserved words. We can group keywords of Java in three separate groups. Some keywords are in use as of today. Some are reserved for future use. Some of them actually represent values. It goes without saying that we cannot use keywords as variables (or user-defined name for any other entity like method).

Table 2.4 illustrates the keywords from Java which are currently in use.

Table 2.4. Keywords in Java

abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
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		

3. Data Types

Basic Data Types:

Primitive Java Data Types

Java supports the following primitive, or basic, data types:

- **`int`**, `long`, or `short` — *to represent integer numbers*
- **`double`** or `float` — *to represent decimal numbers*
- **`char`** — *to represent character values*
- **`boolean`** — *to represent logical values*
- **`void`** — *to represent "no type"*

Each numeric type has a largest and smallest possible value, as indicated in table 1.10.[4]

Most programs use `int` for integers and `double` for decimal numbers, while `long`, `short`, and `float` are needed only in special situations.

Type	Range	
`double`	largest positive/negative value:	±1.7976931348623157E308
	smallest non-zero value:	±4.9E-324
	significant digits:	16 digits after decimal point
`float`	largest positive/negative value:	±3.4028235E38
	smallest non-zero value:	±1.4E-45
	significant digits:	8 digits after decimal point
`int`	largest value	2147483647
	smallest value:	-2147483648
`short`	largest value	32767
	smallest value:	-32768
`long`	largest value	9223372036854775807
	smallest value:	-9223372036854775808

Table: Ranges for valid decimal types

Each type can contain values called literals or unnamed constants in a particular format.

Literals

Literals are constant values for the basic data types. Java supports the following literals:

- `int, short:` *digits only, with possible leading plus (+) or minus (-) sign*
- `long:` *like* `int` *literals, but must end with an* `"L"`
- `double:` *digits including possible periodic point or leading plus (+) or minus (-) sign, or numbers in scientific notation* `#.##############E±###`,[5] *where each* `#` *represents a digit*
- `float:` *like* `double` *literals, but must end with an* `"F"`
- `char:` *Single Unicode characters enclosed in single quotes, including the special control sequences described in table 1.11*[6]
- `boolean:` `true` *or* `false`

In addition, Java has an object literal called `null` *for object references.*

Character literals include the following special characters called control sequences:

Control Sequence	Meaning	Control Sequence	Meaning
\n	new line	\t	tab character
\b	backspace	\r	return
\f	form feed	\\	backslash
\'	single quote	\"	double quote

Table: Common character control sequences

The ranges for the numeric types are the same, regardless of the underlying operating system (after all, programs run under the JVM, not the native operating system). In languages such as C or C++ an integer sometimes has a range similar to a Java `short`, and sometimes that of a Java `int`, depending on the underlying operating system, which can cause different results if the same program runs on different systems.

To use the basic data types to store information, we must define variables that have one of these types:

Declaration of Variables

To declare a variable that can store data of a specific type, the syntax:

```
type varName [, varName2,..., varNameN];
```

[5] For example, the `double` number `1.23456E002 = 1.234562 = 123.456`

[6] Unicode characters support characters in multiple languages and are defined according to their "Unicode Attribute table" (see `http://www.unicode.org/`). Every character on a standard US keyboard is a valid Unicode character.

is used, where `type` is one of the basic data types, `varName` is the name of the variable, and `varName2, ..., varNameN` are optional additional variables of that type. Variables can be declared virtually anywhere in a Java program.

Variables must have a name and there are a few rules to follow when choosing variable names:

Valid Names for Variables

A variable name must start with a letter, a dollar sign '$', or the underscore character '_', followed by any character or number. It can not contain spaces. The reserved keywords listed in the table below can not be used for variable names. Variable names are case-sensitive.

Java Reserved Keywords					
Abstract	boolean	break	byte	case	catch
Char	Class	const	continue	default	do
Double	Else	extends	false	final	finally
Float	For	goto	if	implements	import
instanceof	Int	interface	long	native	new
null	Package	private	protected	public	return
short	Static	super	switch	synchronized	this
throw	Throws	transient	true	try	void
volatile	While				

Table 1.12: Reserved keywords in Java

Example: Declaring variables

Declare one variable each of type `int`, `double`, `char`, and two variables of type `boolean`.

This is an easy example. We declare our variables as follows:

```
int anInteger;
double aDouble;
char aChar;
boolean aBoolean, anotherBoolean;
```

■

Assigning a Value to a Variable

To assign a value to a declared variable, the assignment operator "=" is used:

```
varName = [varName2 = ... ] expression;
```

Assignments are made by first evaluating the expression on right, then assigning the resulting value to the variable on the left. Numeric values of a type with smaller range are compatible with numeric variables of a type with larger range (compare table 13). Variables can be declared and assigned a value in one expression using the syntax:

```
type varName = expression [, varname2 = expression2, ...];
```

The assignment operator looks like the mathematical equal sign, but it is different. For example, as a mathematical expression

$$x = 2x+1$$

is an equation which can be solved for *x*. As Java code, the same expression

```
x = 2*x + 1;
```

means to first evaluate the right side `2*x + 1` by taking the current value of `x`, multiplying it by `2`, and adding `1`. Then the resulting value is stored in `x` (so that now `x` has a new value).

Value and variable types must be compatible with each other, as shown in table 1.13.

Value Type	Compatible Variable Type
double	double
int	int, double
char	char, int, double
boolean	boolean

Table 1.13: Value types compatible to variable types

Example: Declaring variables and assigning values

Declare an `int`, three `double`, one `char`, and one `boolean` variable. Assign to them some suitable values.

There are two possible solutions. Variables can be declared first and a value can be assigned to them at a later time:

```
int anInteger;
double number1, number2, number3;
anInteger = 10;
number1 = number2 = 20.0;
number3 = 30;

char cc;
cc = 'B';
boolean okay;
okay = true;
```

Alternatively, variables can be declared and initialized in one statement:

```
int anInteger = 10;
double number1 = 20.0, number2 = 20.0, number3 = 30;
char cc = 'B';
boolean okay = true;
```

Software Engineering Tip: Variables serve a purpose and the name of a variable should reflect that purpose to improve the readability of your program. Avoid one-letter variable names[7]. Do not reuse a variable whose name does not reflect its purpose.

Whenever possible, assign an initial value to every variable at the time it is declared. If a variable is declared without assigning a value to it, all basic types except `boolean` are automatically set to `0`, `boolean` is set to `false`, and all other types are set to `null`.[8]

[7] If a variable serves a minor role in a code segment (such as a counter in a loop) it can be declared using a one-letter variable name.

[8] The compiler may display an error message if it encounters variables that are not explicitly initialized.

> Declare variables as close as possible to the code where they are used. Do not declare all variables at once at the beginning of a program (or anywhere else).

Example: Using appropriate variable names

The code segment below computes the perimeter of a rectangle and the area of a triangle. Rewrite that segment using more appropriate variable names and compare the readability of both segments:

```
double x, y, z;
x = 10.0;
y = 20.0;
z = 2*(x + y);
w = 0.5 * x * y;
```

This code computes the perimeter `z` of a rectangle with width `x` and length `y` and the area `w` of a triangle with base `x` and height `y`, so the variable names should reflect that. In addition, variables should be assigned a value when they are declared, so the code segment should be rewritten as follows:

```
double width = 10.0, height = 20.0;
double perimeterOfRectangle = 2*(width + height);
double base = width;
double areaOfTriangle = 0.5 * base * height;
```

It is immediately clear that the formulas used are correct. Choosing appropriate variable names clarifies the code significantly and makes it easy to locate potential problems.

□

Basic Arithmetic for Numeric Types

Java support the basic arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder after integer division) for numeric variables and literals. The order of precedence is the standard one from algebra and can be changed using parenthesis.

Each operator has a left and right argument and the type of the result of the computation is determined according to the rules outlined in the table below

Left Argument	Right Argument	Result
int	int	int
int	double	double
double	int	double
double	double	double

Table: Resulting types of the basic arithmetic operations

Example: A Temperature Conversion Program

Create a complete program that converts a temperature from degrees Fahrenheit in degrees Celsius. Use comments to explain your code.

```
public class Converter
{
  public static void main(String args[])
  {
    // Printing out a welcoming message
    System.out.println("Welcome to my Temperature Converter.");
    System.out.println("\nProgram to convert Fahrenheit to Celcius.\n");

    // Defining the temperature value in Fahrenheit
    double temp = 212.0;

    // Applying conversion formula and storing answer in another variable
    double convertedTemp = 5.0 / 9.0 * (temp - 32.0);

    // Printing out the complete answer
    System.out.println(temp + " Fahrenheit = " + convertedTemp + " Celcius.");
    }
}
```

4. Operators

Operator performs operations on operands. Operands may be one, two, or three. Operands are either variables or constants. The operators, which take single operand, are called unary operators. Those taking two are known as binary operators and one having three operands is called ternary operator.

Most of the operators are binary operators, few are unary while there is only one ternary operator (for example "?:" operator).

For further study, we can group operators depending on functionality.

- Arithmetic operators
- Relational (or Conditional) operators
- Shift and Logical operators
- Assignment operators
- Other operators

Let us discuss all of them one by one.

1. Arithmetic operators

From our school days, we are familiar with following arithmetic operators:
addition (+)
subtraction (-)
multiplication (*)
division (/) and
modulo (%).

All these operators are binary operators and work on all integer and floating-point types. Please note that in Java, you can use modulo operator (%) with floating-point operands. (This is not a case with C++.) Following unary operators also work on integers and floating-point numbers:
Unary plus +
Unary minus −
Pre-increment or post-increment ++
Pre-decrement or post-decrement −−

Problem: Write a program to demonstrate use of modulo operator.

Program 5.4. Modulo Operator

```
//      modulo1.java
class  modulo1
{  public static void main(String args[])
       {   int amount = 320;
```

```
            int rupees,paise;
            System.out.println("--->modulo1.java starts<---");
            rupees = amount / 100 ;
            paise = amount % 100 ;
            System.out.println("rupees = " + rupees);
            System.out.println("paise = " + paise);
        }
}
```

If you run the program, you will get the following output.

Output

```
--->modulo1.java starts<---
rupees = 3
paise = 20
```

Table 5.8 illustrates the various relational operators available in Java.

Table 5.8. Relational Operators

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Here we have seen a simple application of modulo operator. It involved integers as operands. The "modulo division" may be applied to floating-point values in Java. This is not permitted in C/C++. However, we have not come across any decent use of this facility.

2. Relational operators

When we use relational operator in an expression it returns Boolean value, that is true or false. Please note that in C++, such expressions actually return number 1 for true and number 0 for false. Hence, a code

```
int k ;
    k = 5>4 ;
```

is valid in C++, but not in Java. Java will flag compilation error saying "Can't convert `boolean` to `int`".

See Program 5.8 `relop1.java` in the end of chapter programs.

3. Logical operators

To work with variables and constants of type `Boolean`, we have logical operators. They are also termed as Boolean operators. Some texts call them conditional operators. Table 5.9 illustrates these operators.

Table 5.9. Logical Operators

Operator	Description
`&&`	Logical AND short-circuit evaluation
`&`	Logical `AND` operator
`| |`	Logical `OR` short-circuit evaluation
`|`	Logical `OR` operator
`!`	Logical negation `(NOT)`
`^`	Exclusive `OR`

Let us study a simple program.

Problem: Write a program to demonstrate evaluation of Boolean expression.

Solution: Let us take the question closest to the heart of first-year students. Whether they will enter second year or not? Consider that there are two semesters whose results are stored as Boolean. Rules of University A are lenient. If a student passes any one semester, he is allowed to go to next year. Rules of University B are stringent. Student must pass both the semesters (see Program 5.5).

Program 5.5. Evaluating Boolean Expression

```
//      logical1.java
class  logical1
{  public static void main(String args[])
       {   boolean semI  = true ;
           boolean semII = false ;
           boolean resultA, resultB ;
           System.out.println("<---logical1--->");
           resultA = semI || semII ;
           resultB = semI && semII ;
           System.out.println("resultA ->" +resultA);
```

```
            System.out.println("resultB ->" +resultB);
        }
}
```

If you run the program, you will get the following output.

Output

```
<---logical1--->
resultA ->true
resultB ->false
```

4. Bit-wise operators

Twenty-five years ago when programming language C was introduced, there was lot of noise regarding bit manipulation. Bit-wise operators were in limelight. However, over the years the enthusiasm has subsided. In the past 5 years, we hardly remember any program written by us using bit manipulation. However, it is still required in specific fields like communication engineering.

Table 5.10 illustrates the bit-wise operators.

Table 5.10. Bit-Wise Operators

Operator	Description
~	Bit-wise (1's) complement
&	Bit-wise AND
^	Bit-wise XOR
|	Bit-wise OR

The following points may be noted regarding use of bit-wise operators:

1. The operands must be of integral (`char`, `byte`, `short`, `int`, or `long`) type. It means `float` and `double` are not allowed as operands.
2. If one of the operand is of type `long`, result is of type `long`.
3. If none of the operand is `long`, result is of type `int`.

Let us see a simple program using bit-wise operators.

Problem: Write a program to demonstrate bit-wise operators.

Solution: In this program we have one example of `AND` and `OR` operator each (see Program 5.6).

Program 5.6. Bit Manipulation

```
//      bit1.java
class  bit1
{  public static void main(String args[])
      {     int i=5;
            long a=2,b ;
```

```
            short c=4,d=3 ;
            System.out.println("<---bit1--->");
             b = i | a ;  // 101 | 010
             d = (short)( d & c ); // 011 &  100
            System.out.println(" ->" + b );
            System.out.println(" ->" + d );
        }
}
```

If you run the program, you will get the following output.

Output

```
<---bit1--->
->7
->0
```

Are the results as expected? You may note that `d & c` produce a result of type `integer`. Hence, typecasting to `short` is required. Effective use of bit-wise operators can be seen in Chapter 24 on "Multimedia Experience".

5. Shift operators

A shift operator also performs bit manipulation by shifting the bits left or right. The bits of the first operand are shifted number of places as specified by the second operand. Java has three operators for this purpose. Table 5.11 summarizes these operators.

Table 5.11. Shift Operators

Operator	Use	Operation
>>	`op1 >> op2`	Shift bits of `op1` right by distance `op2`
<<	`op1 << op2`	Shift bits of `op1` left by distance `op2`
>>>	`op1 >>> op2`	Shift bits of `op1` right by distance `op2` (unsigned)

It goes without saying that these operators work with only operands, which are integral in nature (sorry no floating-point operands).
When we shift right with >> the sign bit (left most) is copied in the left most position. With >>> sign is ignored. Zero is entered in MSB position.
Let us study a problem with shift operators.

Problem: Write a program to study shift operators.

Solution: See Program 5.7.

Program 5.7. Shift Operator

```
//      shift1.java
class  shift1
{  public static void main(String args[])
       {   int i,j;
           byte k = 4 ;
           i =  6 << 2 ;
           j = 23 >> 2;
           k = (byte) (k >> 1)  ;
           System.out.println("--->shift1.java starts<---");
           System.out.println("1) " + i );
           System.out.println("2) " + j );
           System.out.println("3) " + k );
       }
}
```

If you run this program, you will get the following output.

Output

```
--->shift1.java starts<---
1) 24
2) 5
3) 2
```

Note the following:

"<<" is left shift operator. Shifting a binary number by 2 bits left is equal to multiplying by 4(=2*2). Hence, `6 * 4 = 24` is the output.

">>" is right shift operator. Shifting a binary number by 2 bits right is equal to dividing (integer mode) by 4 (= 2 raised to 2). Hence, `23/4 = 5` is the output.

Here 4 is divided by 2. Hence, the answer. The point to note is `k>>1` is evaluated as an integer. Hence, before substituting in `k` we have to typecast it to byte.

6. Assignment operators

We have seen equal sign as basic assignment operator. It assigns value on right-hand side to variable on left-hand side. For example, `marks = 95`.

Many times we need expressions as `marks = marks + 3;` (grace marks). To handle such situations, Java provides shortcut operators. For example, `+=` is a shortcut operator. If you write `marks=+3` it means a shortcut for longer expression `marks = marks+3`. There are many such shortcut operators. They are listed in Table 5.12

Table 5.12. Shortcut Assignment Operators

Operator	Description
=	Simple assignment
*=	Assign product
/=	Assign quotient

Table 5.12. Shortcut Assignment Operators	
Operator	**Description**
%=	Assign remainder (modulus)
+=	Assign sum
-=	Assign difference
&=	Assign bit-wise AND
^=	Assign bit-wise XOR
|=	Assign bit-wise OR
<<=	Assign left shift
>>=	Assign right shift

7. Other (special) operators

Table 5.13 lists other special operators.

Table 5.13. Special Operators	
Operator	**Description**
?:	For conditional evaluation
()	Used in Function call
[]	Used for Array subscript
.	Used as reference to member
new	Used for creating an object or an array
(type)	Used in typecasting the operand
instanceof	Used for checking an object for belonging to a class

8.Conditional operator (? :)

The function of this operator is somewhat similar to `if-then` statement. There is a condition and a selection is based on the evaluation of condition. The only difference is this is an operator and hence returns values. It does not execute different statements. Its syntax is as follows:

```
( condition ) ? A : B
```

The condition is a Boolean expression. If it evaluates to true, value A is returned, otherwise value B is returned. Please note that A and B can be constants or expressions. The brackets

shown around the condition are for readability. They are not compulsory. They may or may not be present.

Problem: Write a program to demonstrate use of conditional operator. Given two array indices, get the value of the array element with higher index.

```
//   cond1.java
//   conditional operator
class cond1
{ public static void main(String args[])
   { int A = 55, B = 65, max,min ;
     System.out.println("<---cond1--->");
     max = A > B ? A :B;
     min = (A<B)? A:B;
     System.out.println("maximum is "+
max);
     System.out.println("minimum is "+
min);
   }
}
```

If you run the program, you will get the following output.

Output

```
<---cond1--->
maximum is 65
minimum is 55
```


9. Operator instanceof

This is a binary operator. It returns a value of type Boolean. If `operand1`, which is an object, is instance of `operand2`, which is a class, the result is true. Else, result is false. At the beginning, one will wonder what the use of such an operator is. When we write a program, it is we who decide variables (objects) names and its types (class). Well this is a general picture. In specific cases, we are allowed to move super-class references to (point to) sub-class object. This creates a possibility that reference of a particular type may or may not be referencing to object of its own class. Therefore, operator instanceof will be helpful to us. Let us write a simple program and get familiar with it.

Problem: Write a program to demonstrate operator `instanceof`.

Solution: We will define a super class and a sub-class. Then we will make reference of super-class po sub-class and check the result

```
//   inst1.java
class time0
      { public int numsec ; // (int n) ;
      } ;
class clock extends time0
       {  String st1;
        } ;
```

```
class inst1
 {  public static void main (String argv[])
    {
    System.out.println("<---inst1.java--->");
    clock clk1 = new clock();
    time0 t1 = new time0();
    time0 t2 ;

    System.out.println("1. t1 instanceof time0 " + ( t1 instanceof
time0) );
    System.out.println("2. t1 instanceof clock " + (t1 instanceof
clock) ) ;
    t2 = t1 ;
    t1= clk1;

    System.out.println("3. t1 instanceof clock " + (t1 instanceof
clock) ) ;

    System.out.println("4. clk1 instanceof clock "
           + (clk1 instanceof clock) ) ;
    System.out.println("5. clk1 instanceof time0 "
           + (clk1 instanceof time0) ) ;
   }
}
```

If you run the program, you will get the following output.

Output

```
<---inst1.java--->
1. t1 instanceof time0 true
2. t1 instanceof clock false
3. t1 instanceof clock true
4. clk1 instanceof clock true
5. clk1 instanceof time0 true
```

10. Operator >>>

Earlier we have studied operator >> It is used for shifting bits right. Shifting right philosophically means dividing by 2. The operator takes care of the sign bit. Digit-wise it means that whatever is earlier leftmost bit, is added as new leftmost bit.
Java has additional operator >>> It works similar to operator >> and shifts the bits in the number to right. It adds zero as leftmost bit. Therefore, its action on positive numbers is identical to operator >>. However, for negative numbers, it gives different result.

Problem: Write a program to demonstrate operator >>>.

```
//   shift3.java
public class shift3
 {
   public static void main(String[] args)
   { int i = 32;
     int j= -32 ;
     int k1,k2,l1,l2;
     System.out.println("<---shift3--->");
     System.out.println("i is " + i ); //Integer.toBinaryString(i));
     System.out.println("j is " + j ); //Integer.toBinaryString(j));
     k1= i>>>2;
     // System.out.println(k1);
     k2= i>>2;
     System.out.println(" using >>> using >>");
     System.out.printf(" i : %10d%10d\n",k1,k2);
     l1=j>>>2;
     //System.out.println(l1);
     l2=j>>2;
     System.out.printf(" j : %10d%10d\n",l1,l2);
  }
}
```

If you run the program, you will get the following output.

Output

```
<---shift3--->
i is 32
j is -32
    using  >>> using >>
i :          8         8
j : 1073741816        -8
```

5. Expressions

We are faced with all sorts of calculations from our school days. Calculations primarily involve numeric expressions. In Java, we will see numeric as well as non-numeric expressions. Expressions are combinations of operators and operands; operands are literals (constants), variables or keywords, or symbols that evaluate a value of some type (for example true). Let us start with arithmetic expressions.

1. Arithmetic expressions
Simple arithmetic expressions can be classified as unary expressions or binary expressions.

2. Arithmetic expressions with single operand
When we apply unary + or − operator to a constant or variable we get unary expression. If we have expression `-Marks`, it is evaluated as `-1 * Marks`. Similarly, expression `+Y` is evaluated as `+1 * Y`.

3. Arithmetic expressions with two operands
These expressions involve two operands and a binary operand. If one fruit costs 3 rupees, five fruits costs
3 * 5 equal to 15 rupees.
Here 3 and 5 are operands and * (for multiplication) is the operator.

4. General arithmetic expressions
General expressions involve many operands and operators. Their evaluation follows standard rules known to us from our school days.

1. When brackets are present, expressions within brackets are evaluated first.
2. Operators with highest precedence are operated before those of lower precedence.
3. With all operators of same precedence, expression is evaluated from left to right.

Associativity
Strictly speaking, when more than one operator with same precedence appears in an expression, we have to look for the associativity of the operators. Most of the operators have left to right associativity. The important exceptions are assignment operator and all of shorthand assignment operators. They operate from right to left.

5. Logical expressions
Expressions involving logical operators are called logical expressions. Alternate name is Boolean expressions. These evaluations follow the Boolean algebra. They result in only true or false answer.
Rules for evaluating such expressions (precedence and left to right) are similar to that of arithmetic expressions.

6. Assignment expressions

Assignment expressions are evaluated as follows. First right-hand side sub-expression is evaluated. Then this result is assigned to variable of left-hand side. It must be noted that at left-hand side we must have something, which can store the value.

`j = 5+5;` is fine, but `3 = 5` is not acceptable. If you write such an expression compiler will generate error message as L-value required.

Java is a strongly typed language. Hence, it does not allow you to assign value of one type to variable of other type in arbitrary manner. Consider following declarations:

```
long n = 2 ;
int j = 5 ;
j = n ;
```

Here `n` is a `long` integer. Its value is assigned to a variable of type `int`. We know that `long` can store values larger than capacity of type `int`. Hence, there is a possibility of error. Hence, Java does not compile this code. It gives error. On the contrary if we write

```
n = j;
```

Java allows this. Reason is very simple. Capacity of `long` is much more than `int`, therefore, `long` can accept any `int` value without error. This is called "widening" principle.

Table 5.1. Assignment Compatibility

Type on left-hand side	Type on right-hand side
short	byte
int	short, byte
long	int, short, byte
float	int, short, byte
double	float, int, short, byte

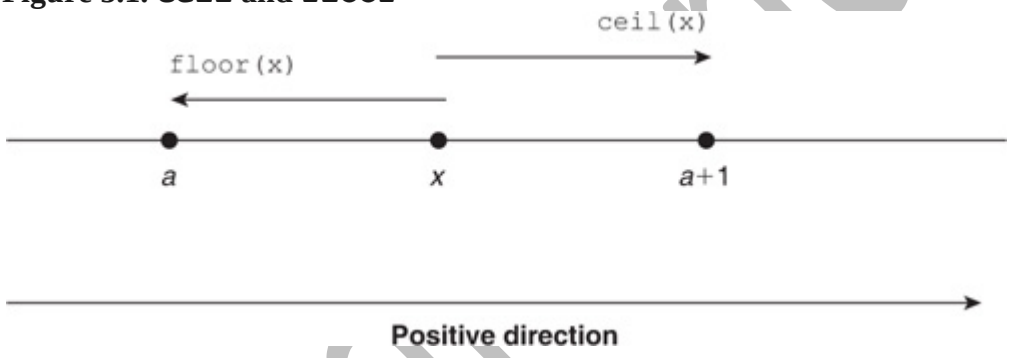
Right from Fortran days, programmers are interested in knowing whether integer values can be put to real variables and vice versa. Let us first talk of substituting integer values in real variables. Type `float` handles values much greater than the largest value of type `integer`. Hence, it can accept integer values. Similarly, `double` can take `long`.

On the other side, conversion from `float` to `int` is not permitted directly. A `float` may contain a fraction, which is not acceptable in `integers`. Hence, specific conversion is required. From our knowledge of school mathematics, we know that there are two well-known ways for conversion. They are truncation and rounding. Java supports method `round()` for rounding. There is no first-hand support for truncation. Java supports methods `ceil()` and `floor()` for this purpose.

Table 5.2. Methods for Converting Real to Integer		
Return type	**Method**	**Description**
`Static double`	`ceil (double x)`	Returns the smallest (closest to negative infinity) `double` value that is, not less than the argument and is equal to a mathematical integer
`Static double`	`floor (double x)`	Returns the largest (closest to positive infinity) `double` value that is, not greater than the argument and is equal to a mathematical integer
`Static long`	`round (double x)`	Returns the closest `long` to the argument
`Static int`	`round (float x)`	Returns the closest `int` to the argument

Figure 5.1 will illustrate their action graphically.

Figure 5.1. `ceil` and `floor`



Please note that in this diagram `a` represents a whole number, which can be either positive or negative. If we want to truncate a real value we can use methods `ceil` and floor (details are left to you as an exercise). Alternately, we can use brute force and use typecasting to truncate a floating-point value. See Program 5.10 `trunc1.java` in the end of chapter programs.

7. Type `char` in assignment

In C language, type `integer` and type `char` are treated as equal and exchangeable. A highly typed Java does not make such assumption. Also, size in bytes of these types also differs. Still Java considers type `char` as something like an (16 bits) `integer`. We are allowed to use operators like (++) with variables of type `char`. As far as assignment is concerned, type `char` can appear on right side of `int` or `float` or `double` assignment expression. However, we cannot assign any non-character variable even of type `byte` to character variable.

Typecasting

All these were natural and logical conversions. What about brute force? What if we want to convert anyway? What if we do not want to care about ensuing error? Well solution in that case is straightforward. We must use a technique called "typecasting".
If we specify a type inside brackets ahead of any expression, then value of that expression is converted to value of type specified in a bracket. For example,

```
(int) 3.5 ;
```

This type of conversion will not alert us even if there is loss of information. Hence, such typecasting must be used with caution.

Keywosrd `Final`

The variables are for holding different values at different times. If we want to keep a value constant, we may declare a variable as final. Such a variable has to be initialized only once. For example,

```
final int j;
    .
    .
    j = 30 ; // first initialization
```

If we try to modify its values at later stage, we will get compilation error.
It is possible to initialize such a variable at the time of declaration only.
For example,

```
final int j= 30 ;
```

```
  final1a.java
class  final1a
{  public static void main(String args[])
       {   int i = 20;
           final int j =30;
           System.out.println("--->final1a.java starts<---");
           i = i + 30 ;
           j = j + 20 ; // note 1
           System.out.println("i = " + i);
           System.out.println("j = " + j);
       }
}
```

If you run the program, you will get the following compilation error.

Output

```
Compiling \JDK6\bin\javac final1a.java
final1a.java:10: cannot assign a value to final variable j
            j = j + 20 ; // note 1
            ^
1 error
press any key to continue
```

Let us remove that line by commenting it out and run the program. In that case, you will get the following output

.

Output

```
--->final1.java starts<---
i = 50
j = 30
```

Please note that use of keyword final is equivalent to const declaration of C++. However, `const` is a reserved word for future use in Java.

6. Control Structures

Simple If Statement

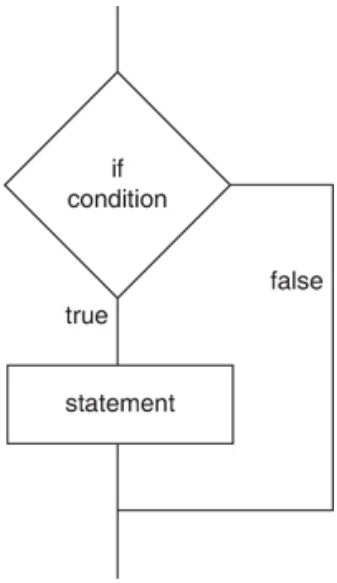
Right from our first exam, we are faced with pass or fail dilemma. If we get minimum marks, then we pass. If we have enough money, we may go to the theatre. The action depends on a condition. This is characterized in `if` statement. Its general syntax is as follows:

```
if ( condition ) statement;
```

It can be described as follows:

- It starts with keyword `if`.
- It is followed by a condition, which is a Boolean expression.
- This condition has to be put in round brackets.
- A statement follows it.
- As usual, semicolon marks the end of statement.
- A block, meaning a group of statements enclosed in curly brackets, is considered as a (single) statement from grammar point of view.

Figure 6.1. `If Statement`



When a computer executes this statement, it first evaluates the condition. If it evaluates to true, computer executes the statement. If the condition fails, no action is taken.

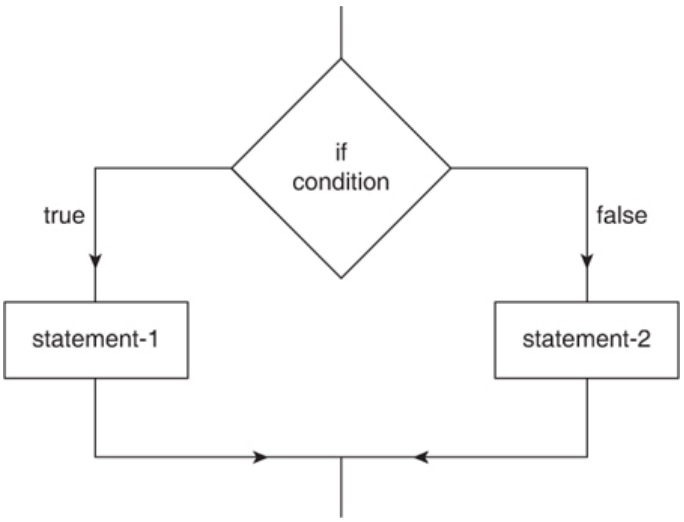
Right from our first exam, we are faced with pass or fail dilemma. If we get more than specified marks, we pass otherwise we fail. There are two different actions, depending on a Boolean condition. This is characterized in `if-else` statement. Its general syntax is as follows:

```
if   ( condition )   statement1 ;
        else   statement2 ;
```


It can be described as follows:

- It starts with keyword `if`.
- It is followed by a condition, which is a Boolean expression.
- This condition has to be put in round brackets.
- It is followed by a statement 1.
- As usual, semicolon marks the end of this statement.
- Next comes keyword `else`.
- It is followed by a statement 2.
- Finally, semicolon marks the end of this statement.
- A block, meaning a group of statements enclosed in curly brackets, is considered as a (single) statement.

Figure 6.2. `If-Else Statement`



When a computer executes this statement, it first evaluates the condition. If it evaluates to true, the computer executes statement1. If the condition fails, it executes statement2.

Problem: Write a program to read marks out of 100 and declare result. The rules are as follows:

60 or more marks	first class
50 to 59 marks	second class
40 to 49 marks	pass class
less than 40 marks	fail

Program 6.1. `If` Statement I

```
//      ifthen1.java
class ifthen1
  { public static void main(String args[])
      {  int marks = 55 ;
         if (marks >=50 )
               if (marks >= 60)
                        System.out.println("First class");
                  else  System.out.println("Second class");

            else
               if (marks >= 40)
                        System.out.println("Pass class");
                  else System.out.println("Fails");
      }
}
```

If you run this program, you will get the following output.

Output

```
Second class
```

Switch-Case Statement

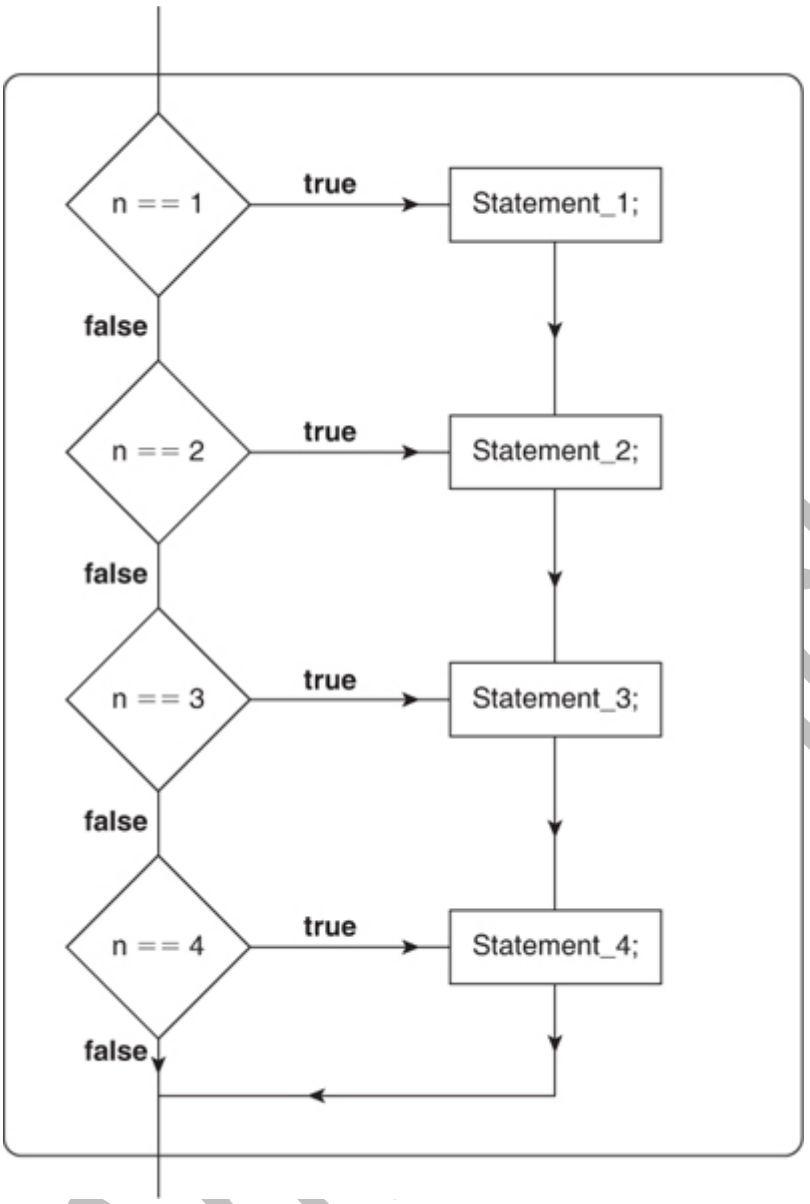
Many times, specific actions take place for specific conditions (cases). Coding becomes complicated on using `if-else` statement. To help the programmers, language supports this new construct, that is, `switch-case` statement. There are some variations in this statement. Hence, let us study it step by step.

In the simplest form, the statement appears as follows.

```
    switch (  n )
       { case 1 : statement_1;
         case 2 : statement_2;
         case 3 : statement_4;
         case 4 : statement_4;
       }

}
```

Figure 6.3 illustrates the logic of this statement.



When control enters the switch construct, it tries to match `n` with values in case clause. If any value matches, then corresponding statement is executed. Thereafter, all the statements (without caring for matching case) are executed. It goes without saying that if there is no match, no statement is executed.

7. Loops

In Java, just like C or C++, there are three structures for looping around a given code. They are

- `for` loop
- `while` loop
- `do-while` loop

Statements `break` and `continue` are used mainly in these structures.

for LOOP

for loop is the most useful statement (facility) in any programming language. A typical statement looks like

```
for(i=0; i<10; i++)
    statement_1;
```

Syntax of for loop can be described as follows:

1. The statement starts with the key word for.
2. Then come three expressions in round bracket.
3. A statement follows it.
4. In the above example, we will call variable i as control variable (cv).
5. First expression initializes cv.
6. Next expression is a condition involving cv.
7. The third expression decides how much cv is incremented (or decremented).

Here statement_1 is executed repeatedly. The action of the for statement can be explained as follows:

1. First control variable i in this statement is initialized, that is i is set to zero.
2. Next statement_1 is executed.
3. Thereafter control variable is incremented, that is i++.
4. Now the condition is tested (i < 10).
5. If the condition is true, statements from step 2 are repeated.
6. Otherwise, the loop terminates and the program goes to the next statement.

Problem: Write a program to find the factorial of a given integer.

Solution: As we are studying for *loop*, we will also show intermediate results in this program. These intermediate results are also factorials themselves (see Program 7.1).

Program 7.1. Factorial I

```
//     factor2.java
class factor2
{ public static void main(String args[])
   { int factorial, n, product;
     int number = 10 ; // given number
     if (number== 1)
           factorial = 1 ;
        else
        { System.out.println("<---factor2.java--->");
          product=1;
          for( n=2; n <= number; n++)
              {  product=product*n ;
System.out.printf("%4d%10d\n",n,product);
               }
          factorial=product;
          System.out.println("");
          System.out.println("The final answer is : "
                             + factorial);
        }
   }
}
```

If you run the program, you will get the following output.

Output

```
<---factor2.java--->
 2         2
 3         6
 4        24
 5       120
 6       720
 7      5040
 8     40320
 9    362880
10   3628800

The final answer is : 3628800
```

while Loop

Sometimes we need to loop around till a particular condition is true. A while statement is very suitable for the purpose. The general form of the while statement is as follows:

```
while ( condition )
  statement_1 ;
```

The statement_1 is repeated while the condition remains true. When the condition becomes false, looping stops. Program control moves ahead. If you want to execute more than one statement, you have to put them in a pair of curly brackets.
Consider a problem in which we want to know the number of years after which the amount grows three times the original. Here, we have to repeat adding interest. We should continue looping while the amount is less than three times its original value. Let us write a program and study it.

Problem: Write a program to find after how many years an amount in fixed deposit becomes three times. Assume the rate of interest as 8%.

Solution: See Program 7.3.

Program 7.3. Fixed Deposit

```
Code View: Scroll / Show All
//     while2.java
class while2
{ public static void main(String args[])
  { double amount = 100000.0;
    double limit = 3 * amount ;
    double rate= 0.08; // rate per unit
    int y = 0 ;
    System.out.println("<---while2.java--->");
    while ( amount < limit)
     {  y = y+1;
        amount = amount * (1+ rate) ;
        System.out.printf("%4d %10.2f\n", y,amount);//Note 1
     }
    System.out.println("\nAfter "+ y + " years amount becomes three times");
  }
}
```

If you run this program, you will get the following output.

Output

```
<---while2.java--->
 1    108000.00
 2    116640.00
```

```
 3    125971.20
 4    136048.90
 5    146932.81
 6    158687.43
 7    171382.43
 8    185093.02
 9    199900.46
10    215892.50
11    233163.90
12    251817.01
13    271962.37
14    293719.36
15    317216.91

After 15 years amount becomes three times
```

do-while Loop

The do-while construct is a slight variation of while construct. A typical structure looks like as follows:

```
do
  statement_1;
while ( condition ) ;
```

When the program enters this loop construct, statements_1 is executed. Then the condition is evaluated. If the condition is true, then control passes to the start of the loop construct, that is keyword do.

Note that a semicolon after statement_1 is mandatory. If you want to repeat more than one statement, put them in a pair of curly brackets.

The most important thing to note is that the body of the loop is executed at least once. In contrast, the while construct body may not be executed at all.

It is understood that if we want to execute more than one statement in the loop, we have to put them in a pair of curly brackets.

In older languages, like Algol and Pascal, a statement called REPEAT UNTIL was present. It was an exact opposite of the while statement. The do-while statement is similar to the repeat until statement. The only difference is that the repeat until statement terminates as the condition becomes true, whereas the do-while terminates when the condition becomes false. Probably developers of C language did not like the repeat until construct. Hence, they replaced it by this do-while construct which is somewhat similar to the while construct.

Let us study a simple program using the do-while statement.

Problem: Write a program to print all natural numbers and their squares such that the square is less than 101. Use do-while construct.

Program 7.5. do-while Loop I

```
//    do1.java
class do1
{ public static void main ( String args[ ] )
  { System.out.println("<---do1.java--->") ;
    int i=1,j=i*i;
    do
     { System.out.printf("%4d %4d\n",i,j);
       i++;
       j= i*i ;
     }
    while(i*i<101);
  }
}
```

If you run the program, you will see the following output.

Output

```
<---do1.java--->
 1    1
 2    4
 3    9
 4   16
 5   25
 6   36
 7   49
 8   64
 9   81
10  100
```

Labelled break and Labelled Continue

1. Labelled break

Earlier we have seen break statement. It helps to jump out of the innermost loop. Sometimes a need may arise that on some condition we would like to jump out not of the innermost loop but out side a deeply nested loop.

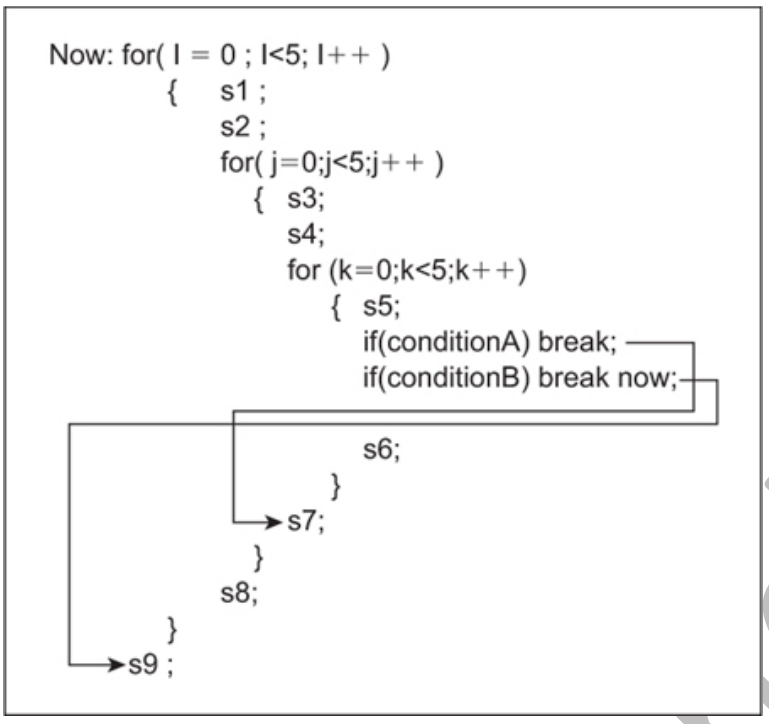
Java being a structured language does not support goto. It supports labelled break instead.

To use this facility, we must assign a label say Now (or any other name) to a loop at any level.

A statement

```
break Now;
```

will move the control out of this labelled loop. Figure 7..1 can explain the action better.



It must be understood that such a labelled break statement must be inside that loop. In Figure 7..1 label is attached to outermost loop. However, label can be attached to any inner loop too.

Problem: Write a program to illustrate labelled break statement.

Solution: Let us take an example of gambling. A man takes some initial amount to Las Vegas. Assume he puts Rs. 10 every time. Most of the times, he wins nothing or a minor amount. Once in a blue moon, he may collect more than Rs. 10,000. In that case, he decides to stop gambling immediately (see Program 7..13).

Program 7..13. Demo of Break

```
Code View: Scroll / Show All
/  break4.java
class break4
  {  public static void main ( String args[ ] )
     {  int amount = 120 ;
        int i,j,k;
        System.out.println("<---break4.java--->" ) ;
        Now: for (j= 0; j<3; j++)
            {   for(i=0;i<4;i++)
                {   amount = amount - 10;
                   k = gamble();
                    amount = amount + k ;
                    if ( k >10000)
```

```
                    { System.out.println("Won " + k +
                        "\n Time to break " );
                      break Now;
                    }
                }
            }
        System.out.println("Out side the Labeled loop" );
        System.out.println("Amount left =" +amount);
    }
   public static int gamble()
    {  return 12000;
    }
}
```

If you run the program, you will get the following output.

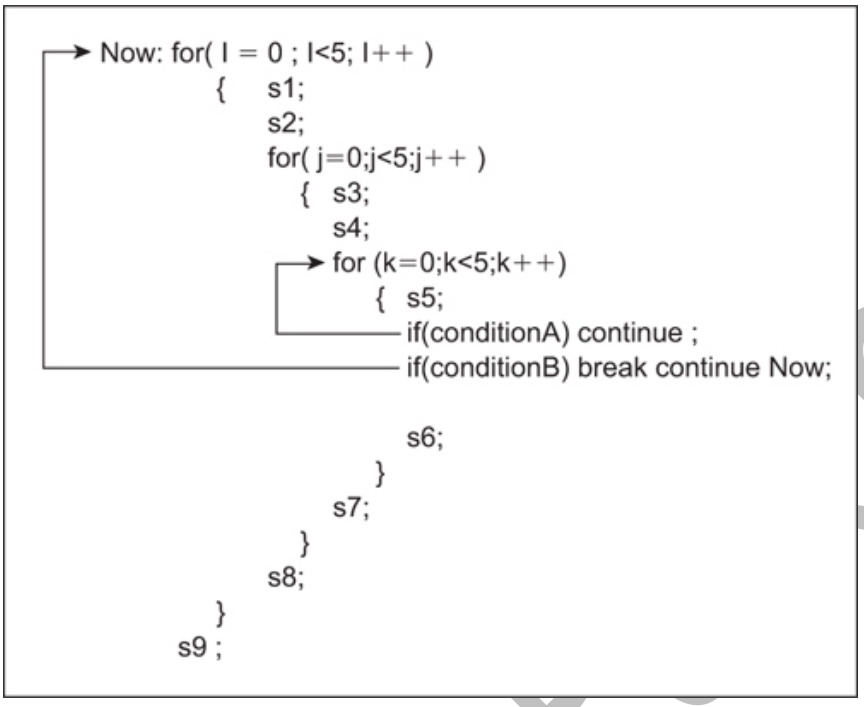
Output

```
<---break4.java--->
Won 12000
Time to break
Out side the Labeled loop
Amount left =12110
```

In the above example, method gamble should contain sophisticated random number generator. This is left to the readers as an exercise.

2. Labelled continue

Just like labelled break, Java provided labelled continue statement. The only difference is that when executed, control goes back to beginning of labelled loop. It means it skips over any unfinished code below it.



As specified, earlier such a break statement must be inside the labelled loop. It cannot be outside the labelled loop. Otherwise there will be a compiler error.

Problem: Write a program to illustrate the use of continue statement with a label.

Solution: In this program, we will set a simple for loop. Inside the loop, we will test marks obtained by the students. If a student gets less than 40 marks, last line in the loop is not printed (see Program 7..14).

Program 7..14. Labelled continue

```
//   continue2.java
class continue2
{  public static void main ( String args[ ] )
    {  int marks[] ={ 55 , 35, 88 , 10 } ;
       System.out.println("--->continue2.java<---") ;
       int i,j;
Label1 : for(j=0;j<2;j++)
        {System.out.println("j= " + j);
         for(i=0;i<4;i++)
          { System.out.println("i= " + i + " Marks =" + marks[i]);
```

```
            if (marks[i] <40) continue Label1;
            System.out.println("---last line in loop " + i );
          }
        }
    }
}
```

If you run the program, you will get the following output.

Output

```
--->continue2.java<---
j= 0
i= 0 Marks =55
---last line in loop 0
i= 1 Marks =35
j= 1
i= 0 Marks =55
---last line in loop 0
i= 1 Marks =35
```

Arrays - String - StringBuffer

1.Arrays

Arrays are generally effective means of storing groups of variables. An array is a group of variables that share the same name and are ordered sequentially from zero to one less than the number of variables in the array. The number of variables that can be stored in an array is called the array's *dimension*. Each variable in the array is called an *element* of the array.

Creating Arrays

There are three steps to creating an array, declaring it, allocating it and initializing it.

Declaring Arrays

Like other variables in Java, an array must have a specific type like byte, int, String or double. Only variables of the appropriate type can be stored in an array. You cannot have an array that will store both ints and Strings, for instance.

Like all other variables in Java an array must be declared. When you declare an array variable you suffix the type with `[]` to indicate that this variable is an array. Here are some examples:


```
int[] k;
float[] yt;
String[] names;
```

In other words you declare an array like you'd declare any other variable except you append brackets to the end of the variable type.

Allocating Arrays

Declaring an array merely says what it is. It does not create the array. To actually create the array (or any other object) use the new operator. When we create an array we need to tell the compiler how many elements will be stored in it. Here's how we'd create the variables declared above: `new`

```
k = new int[3];
yt = new float[7];
names = new String[50];
```

The numbers in the brackets specify the dimension of the array; i.e. how many slots it has to hold values. With the dimensions above k can hold three ints, yt can hold seven floats and names can hold fifty Strings.

Initializing Arrays

Individual elements of the array are referenced by the array name and by an integer which represents their position in the array. The numbers we use to identify them are called subscripts or indexes into the array. Subscripts are consecutive integers beginning with 0. Thus the array k above has elements k[0], k[1], and k[2]. Since we started counting at zero there is no k[3], and trying to access it will generate an ArrayIndexOutOfBoundsException.
subscripts indexes `k k[0] k[1] k[2] k[3] ArrayIndexOutOfBoundsException`

You can use array elements wherever you'd use a similarly typed variable that wasn't part of an array.

Here's how we'd store values in the arrays we've been working with:

```
k[0] = 2;
k[1] = 5;
k[2] = -2;
yt[6] = 7.5f;
names[4] = "Fred";
```

This step is called initializing the array or, more precisely, initializing the elements of the array. Sometimes the phrase "initializing the array" would be reserved for when we initialize all the elements of the array.

For even medium sized arrays, it's unwieldy to specify each element individually. It is often helpful to use `for` loops to initialize the array. For instance here is a loop that fills an array with the squares of the numbers from 0 to 100.

```
float[] squares = new float[101];

for (int i=0; i <= 500; i++) {
  squares[i] = i*2;
}
```

Shortcuts

We can declare and allocate an array at the same time like this:

```
int[] k = new int[3];
float[] yt = new float[7];
String[] names = new String[50];
```

We can even declare, allocate, and initialize an array at the same time providing a list of the initial values inside brackets like so:

```
int[] k = {1, 2, 3};
float[] yt = {0.0f, 1.2f, 3.4f, -9.87f, 65.4f, 0.0f, 567.9f};
```

Two Dimensional Arrays

Declaring, Allocating and Initializing Two Dimensional Arrays

Two dimensional arrays are declared, allocated and initialized much like one dimensional arrays. However we have to specify two dimensions rather than one, and we typically use two nested for loops to fill the array. `for`

The array examples above are filled with the sum of their row and column indices. Here's some code that would create and fill such an array:

```
class FillArray {

  public static void main (String args[]) {

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

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 5; col++) {
        M[row][col] = row+col;
      }
    }

  }

}
```

In two-dimensional arrays ArrayIndexOutOfBounds errors occur whenever you exceed the maximum column index or row index. Unlike two-dimensional C arrays, two-dimensional Java arrays are not just one-dimensional arrays indexed in a funny way.

Multidimensional Arrays

You don't have to stop with two dimensional arrays. Java lets you have arrays of three, four or more dimensions. However chances are pretty good that if you need more than three dimensions in an array, you're probably using the wrong data structure. Even three dimensional arrays are exceptionally rare outside of scientific and engineering applications.

The syntax for three dimensional arrays is a direct extension of that for two-dimensional arrays. Here's a program that declares, allocates and initializes a three-dimensional array:

```
class Fill3DArray {

  public static void main (String args[]) {

    int[][][] M;
    M = new int[4][5][3];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 5; col++) {
        for (int ver=0; ver < 3; ver++) {
          M[row][col][ver] = row+col+ver;
        }
      }
    }

  }

}
```

Example 1 : declaring and initializing 1-dimensional arrays

An array groups elements of the same type. It makes it easy to manipulate the information contained in them.

```
class Arrays1{

public static void main(String args[]){

// this declares an array named x with the type "array of int" and of
// size 10, meaning 10 elements, x[0], x[1] , ... , x[9] ; the first term
// is x[0] and the last term x[9] NOT x[10].
int x[] = new int[10];

// print out the values of x[i] and they are all equal to 0.
for(int i=0; i<=9; i++)
System.out.println("x["+i+"] = "+x[i]);

// assign values to x[i]
for(int i=0; i<=9; i++)
x[i] = i; // for example

// print the assigned values of x[i] : 1,2,.....,9
for(int i=0; i<=9; i++)
System.out.println("x["+i+"] = "+x[i]);

// this declares an array named st the type "array of String"
// and initializes it
String st[]={"first","second","third"};
```

```
// print out st[i]
for(int i=0; i<=2; i++)
System.out.println("st["+i+"] = "+st[i]);

}
}
```

Example 2 : Find the sum of the numbers 2.5, 4.5, 8.9, 5.0 and 8.9

```
class Arrays2{

public static void main(String args[]){

// this declares an array named fl with the type "array of int" and
// initialize its elements

float fl[] = {2.5f, 4.5f, 8.9f, 5.0f, 8.9f};

// find the sum by adding all elements of the array fl
float sum = 0.0f;
for(int i=0; i<= 4; i++)
sum = sum + fl[i];

// displays the sum
System.out.println("sum = "+sum);
}
}
```

Check that the sum displayed is 29.8.

Example 3 : declaring and initializing 2-dimensional arrays

```
class Arrays3{

public static void main(String args[]){

// this declares a 2-dimensional array named x[i][j] of size 4 (4 elements)
// its elements are x[0][0], x[0][1], x[1][0] and x[1][1].
// the first index i indicates the row and the second index indicates the
// column if you think of this array as a matrix.


int x[][] = new int[2][2];

// print out the values of x[i][j] and they are all equal to 0.0.
```

```
for(int i=0; i<=1; i++)
for(int j=0; j<=1; j++)
System.out.println("x["+i+","+j+"] = "+x[i][j]);

// assign values to x[i]
for(int i=0; i<=1; i++)
for(int j=0; j<=1; j++)
x[i][j] = i+j; // for example

// print the assigned values to x[i][j]
for(int i=0; i<=1; i++)
for(int j=0; j<=1; j++)
System.out.println("x["+i+","+j+"] = "+x[i][j]);

// this declares a 2-dimensional array of type String
// and initializes it
String st[][]={{"row 0 column 0","row 0 column 1"}, // first row
{"row 1 column 0","row 1 column 1"}}; // second row

// print out st[i]
for(int i=0; i<=1; i++)
for(int j=0; j<=1; j++)
System.out.println("st["+i+","+j+"] = "+st[i][j]);

}
}
```


2.Strings Class

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are 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 eleven 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 would 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.

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.

After the following two lines of code have been executed, len equals 17:

```
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 would 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 would 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:

SN	Methods with Description
1	char charAt(int index) Returns the character at the specified index.
2	int compareTo(Object o) Compares this String to another Object.
3	int compareTo(String anotherString) Compares two strings lexicographically.
4	int compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
5	String concat(String str) Concatenates the specified string to the end of this string.
6	boolean contentEquals(StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	static String copyValueOf(char[] data) Returns a String that represents the character sequence in the array specified.
8	static String copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
9	boolean endsWith(String suffix) Tests if this string ends with the specified suffix.
10	boolean equals(Object anObject) Compares this string to the specified object.
11	boolean equalsIgnoreCase(String anotherString)

	Compares this String to another String, ignoring case considerations.
12	byte getBytes() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	byte[] getBytes(String charsetName Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
15	int hashCode() Returns a hash code for this string.
16	int indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.
17	int indexOf(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	int indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
19	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
20	String intern() Returns a canonical representation for the string object.
21	int lastIndexOf(int ch) Returns the index within this string of the last occurrence of the specified character.
22	int lastIndexOf(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	int lastIndexOf(String str)

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

35	boolean startsWith(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index.
36	CharSequence subSequence(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
37	String substring(int beginIndex) Returns a new string that is a substring of this string.
38	String substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
39	char[] toCharArray() Converts this string to a new character array.
40	String toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale.
41	String toLowerCase(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.
42	String toString() This object (which is already a string!) is itself returned.
43	String toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale.
44	String toUpperCase(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
45	String trim() Returns a copy of the string, with leading and trailing whitespace omitted.
46	static String valueOf(primitive data type x) Returns the string representation of the passed data type argument.

3.StringBuffer class

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer class

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

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

What is mutable string

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

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```


3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```

5) StringBuffer reverse() method

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

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }
```

6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.


```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. }
10. }
```

7) StringBuffer ensureCapacity() method

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

```
1. class A{
```

```
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }
```

4.StringBuilder class

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

Important Constructors of StringBuilder class

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

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public StringBuilder insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.

public StringBuilder delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	is used to reverse the string.
public int capacity()	is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	is used to return the character at the specified position.
public int length()	is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

```
1. class A{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

```
1. class A{
```

```
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class A{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }
```

4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.


```
1. class A{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```

5) StringBuilder reverse() method

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

```
1. class A{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }
```

6) StringBuilder capacity() method

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1. class A{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. }
10. }
```

7) StringBuilder ensureCapacity() method

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

```
1. class A{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

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

1.Object and Class in Java

Object is the physical as well as logical entity whereas class is the logical entity only.

Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Class in Java

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Syntax to declare a class:

```
1. class <class_name>{
2.     data member;
3.     method;
4. }
```

Simple Example of Object and Class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.


```
1. class Student1{
2.  int id;//data member (also instance variable)
3.  String name;//data member(also instance variable)
4.
5.  public static void main(String args[]){
6.   Student1 s1=new Student1();//creating an object of Student
7.   System.out.println(s1.id);
8.   System.out.println(s1.name);
9.  }
10. }
```

```
Output:0 null
```

Instance variable in Java

A variable that is created inside the class but outside the method, is known as instance variable.Instance variable doesn't get memory at compile time.It gets memory at runtime when object(instance) is created.That is why, it is known as instance variable.

Method in Java

In java, a method is like function i.e. used to expose behaviour of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword

The new keyword is used to allocate memory at runtime.

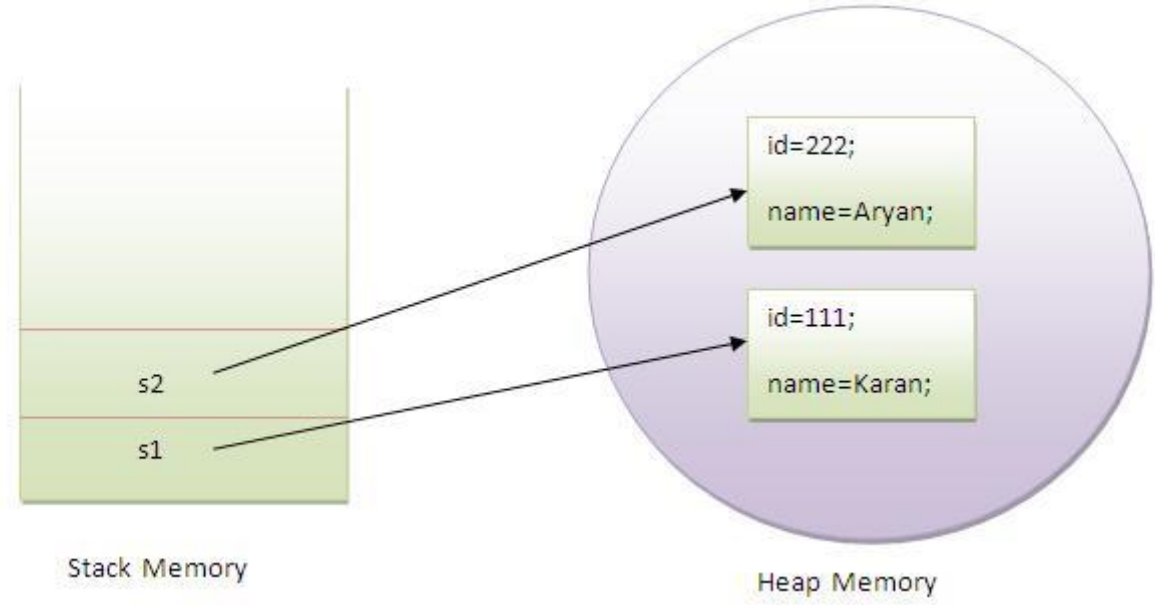
Example of Object and class that maintains the records of students

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method on it. Here, we are displaying the state (data) of the objects by invoking the displayInformation method.

```
1.  class Student2{
2.   int rollno;
3.   String name;
4.
5.   void insertRecord(int r, String n){  //method
6.    rollno=r;
7.    name=n;
8.   }
9.
10.  void displayInformation(){System.out.println(rollno+" "+name);}//method
11.
12.  public static void main(String args[]){
13.   Student2 s1=new Student2();
14.   Student2 s2=new Student2();
15.
16.   s1.insertRecord(111,"Karan");
17.   s2.insertRecord(222,"Aryan");
18.
19.   s1.displayInformation();
20.   s2.displayInformation();
21.
22.  }
23. }
```

```
111 Karan
```

```
222 Aryan
```



As you see in the above figure, object gets the memory in Heap area and reference variable refers to the object allocated in the Heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Another Example of Object and Class

There is given another example that maintains the records of Rectangle class. Its exaplanation is same as in the above Student class example.

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.
5.   void insert(int l,int w){
6.    length=l;
7.    width=w;
8.   }
9.
10.  void calculateArea(){System.out.println(length*width);}
11.
12.  public static void main(String args[]){
13.   Rectangle r1=new Rectangle();
14.   Rectangle r2=new Rectangle();
15.
16.   r1.insert(11,5);
```

```
17.   r2.insert(3,15);
18.
19.   r1.calculateArea();
20.   r2.calculateArea();
21. }
22. }
```

```
Output:55
       45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By factory method etc.

Annonymous object

Annonymous simply means nameless.An object that have no reference is known as annonymous object.

If you have to use an object only once, annonymous object is a good approach.


```
1.   class Calculation{
2.
3.    void fact(int  n){
4.     int fact=1;
5.     for(int i=1;i<=n;i++){
6.      fact=fact*i;
7.     }
8.    System.out.println("factorial is "+fact);
9.   }
10.
11. public static void main(String args[]){
12.  new Calculation().fact(5);//calling method with annonymous object
13. }
14. }
```

```
Output:Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

1. Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects

Let's see the example:

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.
5.   void insert(int l,int w){
6.    length=l;
7.    width=w;
8.   }
9.
10.  void calculateArea(){System.out.println(length*width);}
11.
12.  public static void main(String args[]){
13.   Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
14.
15.   r1.insert(11,5);
16.   r2.insert(3,15);
17.
18.   r1.calculateArea();
19.   r2.calculateArea();
20. }
21. }
```

```
Output:55
```

2.Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand

the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading?

Method overloading **increases the readability of the program**.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1)Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
1. class Calculation{
2.   void sum(int a,int b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.   public static void main(String args[]){
6.   Calculation obj=new Calculation();
7.   obj.sum(10,10,10);
8.   obj.sum(20,20);
9.
10.  }
11. }
```

2)Example of Method Overloading by changing data type of argument

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
1. class Calculation2{
2.   void sum(int a,int b){System.out.println(a+b);}
3.   void sum(double a,double b){System.out.println(a+b);}
4.
```

```
5.  public static void main(String args[]){
6.  Calculation2 obj=new Calculation2();
7.  obj.sum(10.5,10.5);
8.  obj.sum(20,20);
9.
10.  }
11. }
```

```
Output:21.0
       40
```

```
Output:30
       40
```

Que) Why Method Overloaing is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```
1.  class Calculation3{
2.    int sum(int a,int b){System.out.println(a+b);}
3.    double sum(int a,int b){System.out.println(a+b);}
4.
5.    public static void main(String args[]){
6.    Calculation3 obj=new Calculation3();
7.    int result=obj.sum(20,20); //Compile Time Error
8.
9.    }
10. }
```

int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

---

Can we overload main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

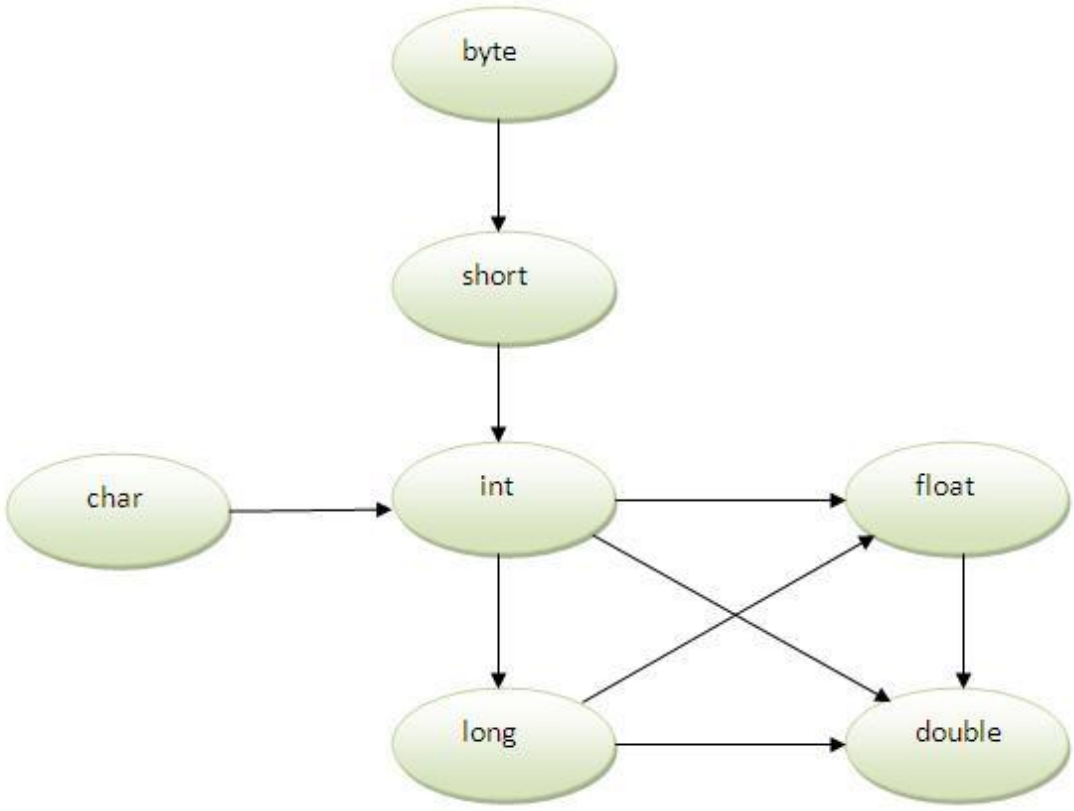
```
1.  class Overloading1{
2.    public static void main(int a){
3.    System.out.println(a);
```

```
4.    }
5.
6.    public static void main(String args[]){
7.    System.out.println("main() method invoked");
8.    main(10);
9.    }
10. }
```

```
Output:main() method invoked
       10
```

Method Overloading and TypePromotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1. class OverloadingCalculation1{
2.   void sum(int a,long b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation1 obj=new OverloadingCalculation1();
7.   obj.sum(20,20);//now second int literal will be promoted to long
8.   obj.sum(20,20,20);
9.
10.  }
11. }
```

```
Output:40
       60
```

Example of Method Overloading with TypePromotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1. class OverloadingCalculation2{
2.   void sum(int a,int b){System.out.println("int arg method invoked");}
3.   void sum(long a,long b){System.out.println("long arg method invoked");}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation2 obj=new OverloadingCalculation2();
7.   obj.sum(20,20);//now int arg sum() method gets invoked
8.   }
9.  }
```

```
Output:int arg method invoked
```

Example of Method Overloading with TypePromotion in case ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
1. class OverloadingCalculation3{
2.   void sum(int a,long b){System.out.println("a method invoked");}
3.   void sum(long a,int b){System.out.println("b method invoked");}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation3 obj=new OverloadingCalculation3();
7.   obj.sum(20,20);//now ambiguity
```

```
8.    }
9.  }
```

```
Output:Compile Time Error
```

One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitely

3.Constructor in Java

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

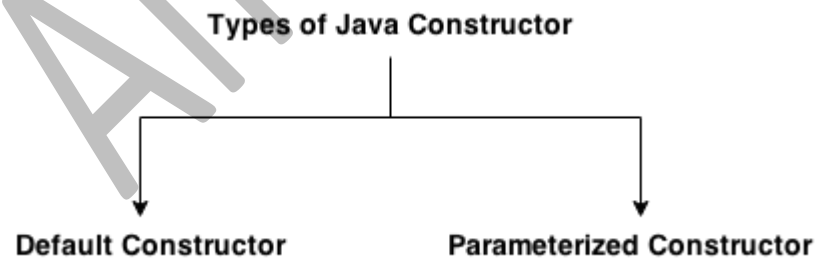
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

```
1. <class_name>(){}
```

Example of default constructor

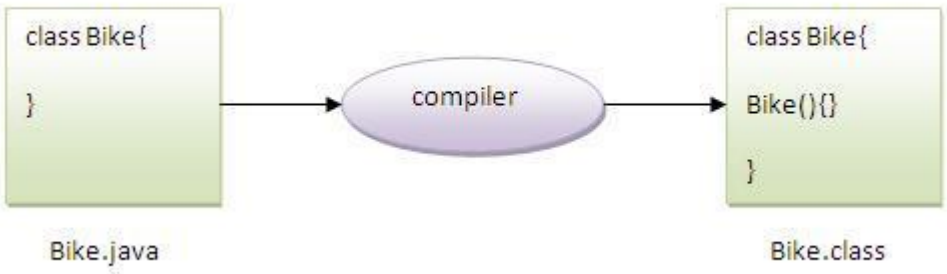
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1. class Bike1{
2. Bike1(){System.out.println("Bike is created");}
3. public static void main(String args[]){
4. Bike1 b=new Bike1();
5. }
6. }
```

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of default constructor?

Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
1. class Student3{
2. int id;
3. String name;
4.
5. void display(){System.out.println(id+" "+name);}
6.
7. public static void main(String args[]){
8. Student3 s1=new Student3();
9. Student3 s2=new Student3();
10. s1.display();
11. s2.display();
12. }
13. }
```

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor.

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1. class Student4{
2.     int id;
3.     String name;
4.
5.     Student4(int i,String n){
```

```
6.      id = i;
7.      name = n;
8.      }
9.      void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.     Student4 s1 = new Student4(111,"Karan");
13.     Student4 s2 = new Student4(222,"Aryan");
14.     s1.display();
15.     s2.display();
16.    }
17. }
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
1.  class Student5{
2.      int id;
3.      String name;
4.      int age;
5.      Student5(int i,String n){
6.      id = i;
7.      name = n;
8.      }
9.      Student5(int i,String n,int a){
10.     id = i;
11.     name = n;
12.     age=a;
13.     }
14.     void display(){System.out.println(id+" "+name+" "+age);}
15.
16.     public static void main(String args[]){
17.     Student5 s1 = new Student5(111,"Karan");
18.     Student5 s2 = new Student5(222,"Aryan",25);
19.     s1.display();
```

```
20.     s2.display();
21.    }
22. }
```

Output:

```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
1.  class Student6{
2.      int id;
3.      String name;
4.      Student6(int i,String n){
5.      id = i;
6.      name = n;
7.      }
8.
9.      Student6(Student6 s){
10.     id = s.id;
11.     name =s.name;
12.     }
13.     void display(){System.out.println(id+" "+name);}
14.
15.     public static void main(String args[]){
16.     Student6 s1 = new Student6(111,"Karan");
17.     Student6 s2 = new Student6(s1);
18.     s1.display();
19.     s2.display();
20.    }
21. }
```

Output:

```
111 Karan
111 Karan
```


Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.  class Student7{
2.      int id;
3.      String name;
4.      Student7(int i,String n){
5.      id = i;
6.      name = n;
7.      }
8.      Student7(){}
9.      void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.     Student7 s1 = new Student7(111,"Karan");
13.     Student7 s2 = new Student7();
14.     s2.id=s1.id;
```

```
15.     s2.name=s1.name;
16.     s1.display();
17.     s2.display();
18.    }
19. }
```

Output:

```
111 Karan
111 Karan
```

Q) Does constructor return any value?

Ans:yes, that is current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

4.Java static keyword

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

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

---

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

```
1. class Student{
2.     int rollno;
3.     String name;
4.     String college="ITS";
5. }
```

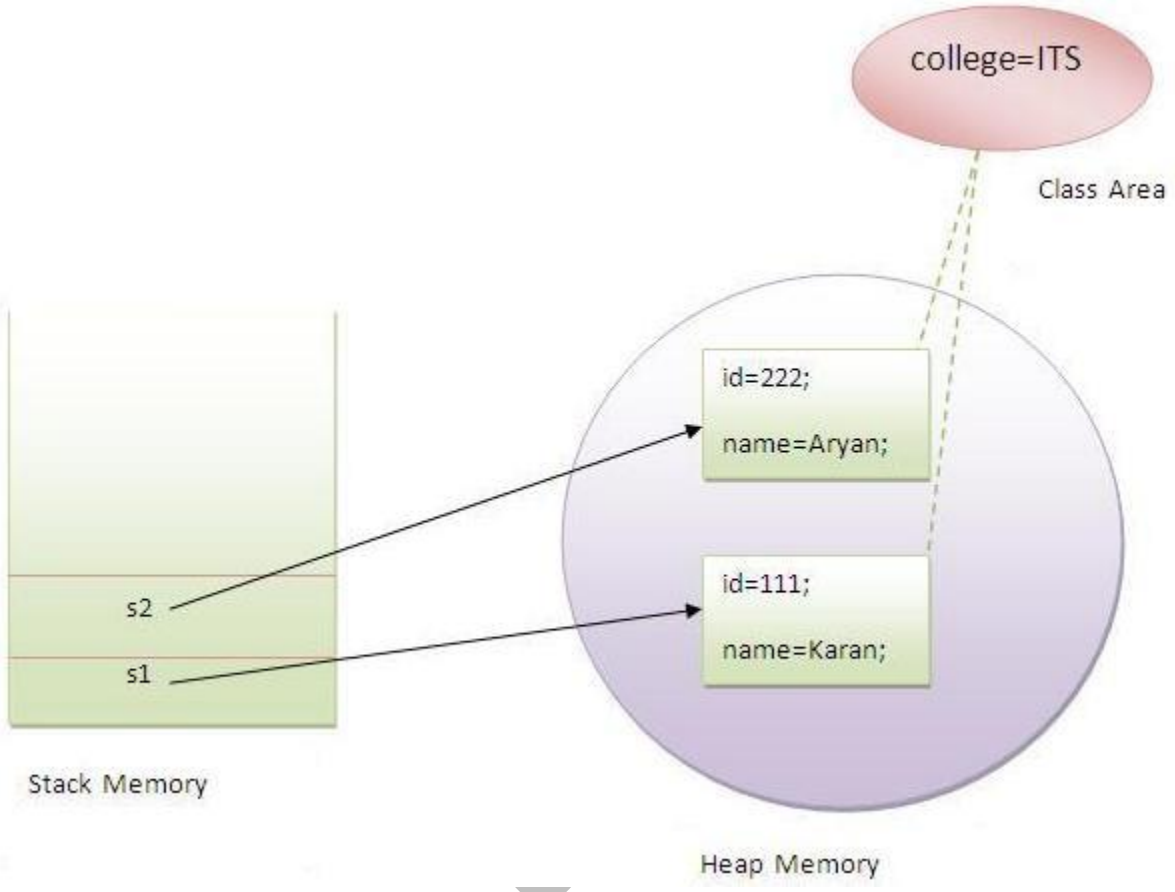
Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

Java static property is shared to all objects.

Example of static variable

```
1.  //Program of static variable
2.
3.  class Student8{
4.    int rollno;
5.    String name;
6.    static String college ="ITS";
7.
8.    Student8(int r,String n){
9.    rollno = r;
10.   name = n;
11.   }
12. void display (){System.out.println(rollno+" "+name+" "+college);}
13.
14. public static void main(String args[]){
15. Student8 s1 = new Student8(111,"Karan");
16. Student8 s2 = new Student8(222,"Aryan");
17.
18. s1.display();
19. s2.display();
20. }
21. }
```

```
Output:111 Karan ITS
       222 Aryan ITS
```



Program of counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
1. class Counter{
2. int count=0;//will get memory when instance is created
3.
4. Counter(){
5. count++;
6. System.out.println(count);
7. }
8.
9. public static void main(String args[]){
10.
11. Counter c1=new Counter();
12. Counter c2=new Counter();
```

```
13. Counter c3=new Counter();
14.
15.  }
16. }
```

```
Output:1
       1
       1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1.  class Counter2{
2.  static int count=0;//will get memory only once and retain its value
3.
4.  Counter2(){
5.  count++;
6.  System.out.println(count);
7.  }
8.
9.  public static void main(String args[]){
10.
11. Counter2 c1=new Counter2();
12. Counter2 c2=new Counter2();
13. Counter2 c3=new Counter2();
14.
15.  }
16. }
```

```
Output:1
       2
       3
```


---

2) Java static method

If you apply static keyword with any method, it is known as static method.

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

Example of static method

```
1.  //Program of changing the common property of all objects(static field).
2.
3.  class Student9{
4.      int rollno;
5.      String name;
6.      static String college = "ITS";
7.
8.      static void change(){
9.      college = "BBDIT";
10.     }
11.
12.     Student9(int r, String n){
13.     rollno = r;
14.     name = n;
15.     }
16.
17.     void display (){System.out.println(rollno+" "+name+" "+college);}
18.
19.    public static void main(String args[]){
20.    Student9.change();
21.
22.    Student9 s1 = new Student9 (111,"Karan");
23.    Student9 s2 = new Student9 (222,"Aryan");
24.    Student9 s3 = new Student9 (333,"Sonoo");
25.
26.    s1.display();
27.    s2.display();
28.    s3.display();
29.    }
30. }
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```


Another example of static method that performs normal calculation

```
1.  //Program to get cube of a given number by static method
2.
3.  class Calculate{
4.    static int cube(int x){
5.    return x*x*x;
```

```
6.    }
7.
8.    public static void main(String args[]){
9.    int result=Calculate.cube(5);
10.   System.out.println(result);
11.   }
12. }
```

Output:125

Restrictions for static method

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

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

```
1.  class A{
2.   int a=40;//non static
3.
4.   public static void main(String args[]){
5.    System.out.println(a);
6.   }
7.  }
```

Output:Compile Time Error

Q) why java main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

---

3) Java static block

- Is used to initialize the static data member.

- It is executed before main method at the time of classloading.

Example of static block

```
1. class A2{
2.   static{System.out.println("static block is invoked");}
3.   public static void main(String args[]){
4.    System.out.println("Hello main");
5.   }
6. }
```

```
Output:static block is invoked
       Hello main
```

Q) Can we execute a program without main() method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```
1. class A3{
2.   static{
3.   System.out.println("static block is invoked");
4.   System.exit(0);
5.   }
6. }
```



```
Output:static block is invoked (if not JDK7)
```

In JDK7 and above, output will be:

```
Output:Error: Main method not found in class A3, please define the main
method as:
public static void main(String[] args)
```

5.this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

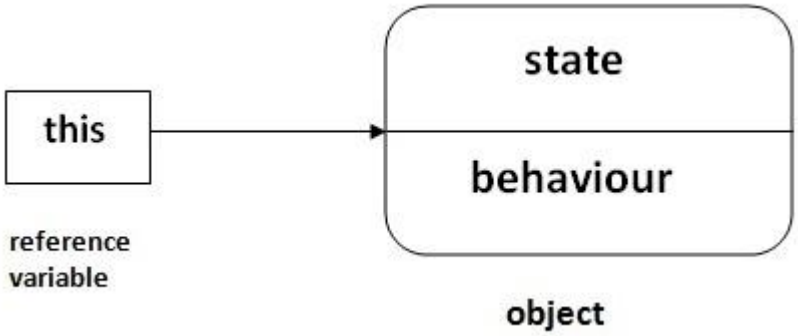
Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.

2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.



1) The this keyword can be used to refer current class instance variable.

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

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
1.  class Student10{
2.      int id;
3.      String name;
4.
5.      Student10(int id,String name){
6.      id = id;
7.      name = name;
8.      }
9.      void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.     Student10 s1 = new Student10(111,"Karan");
13.     Student10 s2 = new Student10(321,"Aryan");
```

```
14.    s1.display();
15.    s2.display();
16.    }
17. }
```

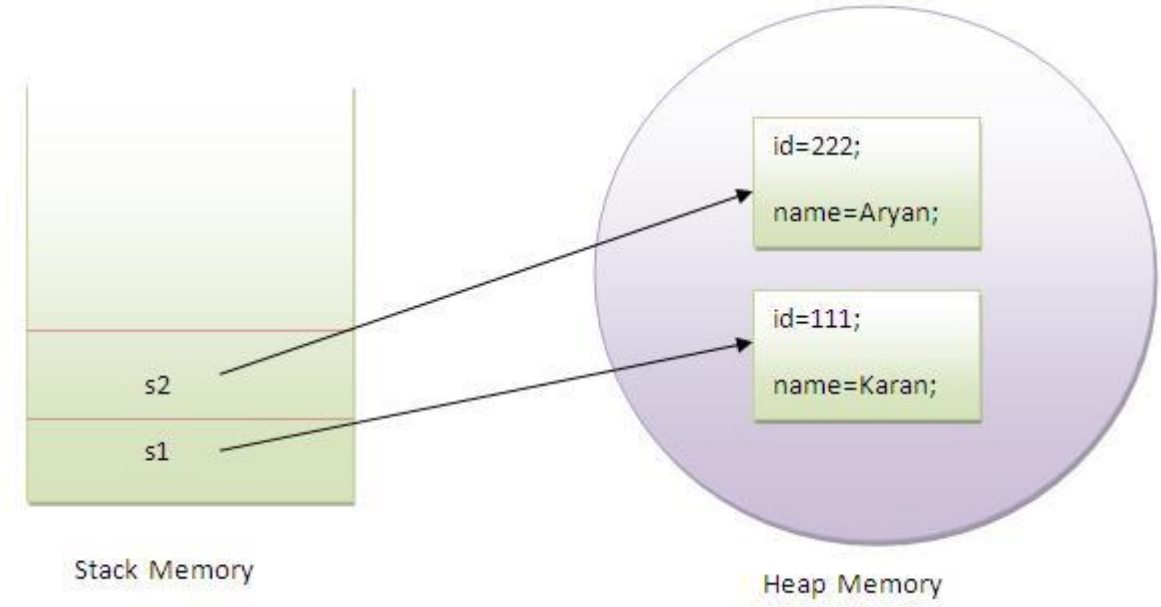
```
Output:0 null
       0 null
```

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
1.  //example of this keyword
2.  class Student11{
3.     int id;
4.     String name;
5.
6.     Student11(int id,String name){
7.     this.id = id;
8.     this.name = name;
9.     }
10.    void display(){System.out.println(id+" "+name);}
11.    public static void main(String args[]){
12.    Student11 s1 = new Student11(111,"Karan");
13.    Student11 s2 = new Student11(222,"Aryan");
14.    s1.display();
15.    s2.display();
16. }
17. }
```

```
Output111 Karan
      222 Aryan
```



If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required


```
1.  class Student12{
2.      int id;
3.      String name;
4.
5.      Student12(int i,String n){
6.      id = i;
7.      name = n;
8.      }
9.      void display(){System.out.println(id+" "+name);}
10.     public static void main(String args[]){
11.     Student12 e1 = new Student12(111,"karan");
12.     Student12 e2 = new Student12(222,"Aryan");
13.     e1.display();
14.     e2.display();
15. }
16. }
```

```
Output:111 Karan
       222 Aryan
```

2) this() can be used to invoked current class constructor.

The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

```
1.  //Program of this() constructor call (constructor chaining)
2.
3.  class Student13{
4.     int id;
5.     String name;
6.     Student13(){System.out.println("default constructor is invoked");}
7.
8.     Student13(int id,String name){
9.     this ();//it is used to invoked current class constructor.
10.    this.id = id;
11.    this.name = name;
12.    }
13.    void display(){System.out.println(id+" "+name);}
14.
15.    public static void main(String args[]){
16.    Student13 e1 = new Student13(111,"karan");
17.    Student13 e2 = new Student13(222,"Aryan");
18.    e1.display();
19.    e2.display();
20.   }
21. }
```

```
Output:
       default constructor is invoked
       default constructor is invoked
       111 Karan
       222 Aryan
```

Where to use this() constructor call?

The this() constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
1.  class Student14{
2.     int id;
3.     String name;
4.     String city;
5.
```

```
6.      Student14(int id,String name){
7.      this.id = id;
8.      this.name = name;
9.      }
10.     Student14(int id,String name,String city){
11.     this(id,name);//now no need to initialize id and name
12.     this.city=city;
13.     }
14.     void display(){System.out.println(id+" "+name+" "+city);}
15.
16.     public static void main(String args[]){
17.     Student14 e1 = new Student14(111,"karan");
18.     Student14 e2 = new Student14(222,"Aryan","delhi");
19.     e1.display();
20.     e2.display();
21.    }
22. }
```

```
Output:111 Karan null
       222 Aryan delhi
```

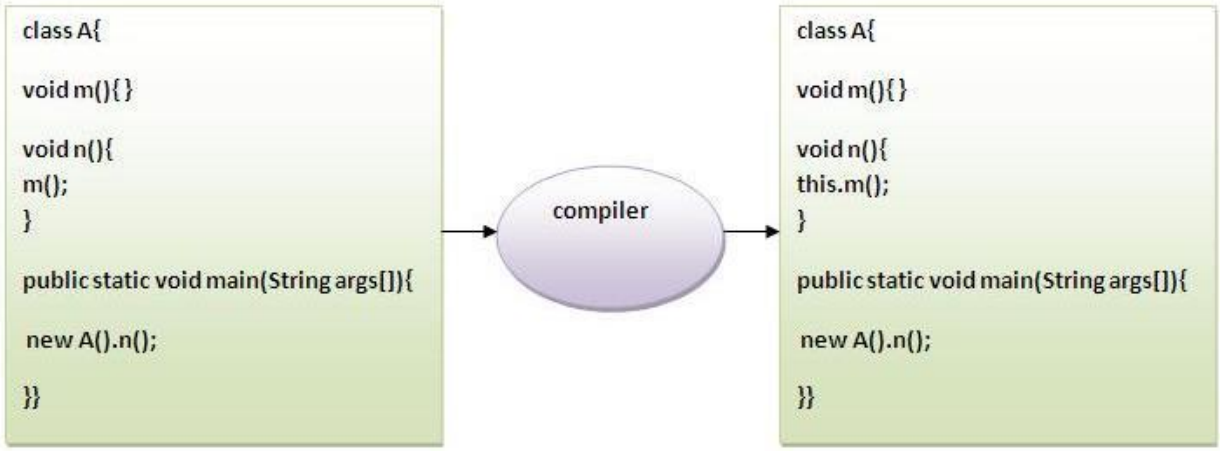
Rule: Call to this() must be the first statement in constructor.

```
1.   class Student15{
2.      int id;
3.      String name;
4.      Student15(){System.out.println("default constructor is invoked");}
5.
6.      Student15(int id,String name){
7.      id = id;
8.      name = name;
9.      this ();//must be the first statement
10.     }
11.     void display(){System.out.println(id+" "+name);}
12.
13.     public static void main(String args[]){
14.     Student15 e1 = new Student15(111,"karan");
15.     Student15 e2 = new Student15(222,"Aryan");
16.     e1.display();
17.     e2.display();
18.    }
19. }
```

```
Output:Compile Time Error
```

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example




```
1.  class S{
2.    void m(){
3.    System.out.println("method is invoked");
4.    }
5.    void n(){
6.    this.m();//no need because compiler does it for you.
7.    }
8.    void p(){
9.    n();//complier will add this to invoke n() method as this.n()
10.   }
11.   public static void main(String args[]){
12.   S s1 = new S();
13.   s1.p();
14.   }
15. }
```

```
Output:method is invoked
```

---

4) this keyword can be passed as an argument in the method.

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
1.  class S2{
2.    void m(S2 obj){
3.    System.out.println("method is invoked");
4.    }
5.    void p(){
6.    m(this);
7.    }
8.
9.    public static void main(String args[]){
10.   S2 s1 = new S2();
11.   s1.p();
12.   }
13. }
```

```
Output:method is invoked
```

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one.

---

5) The this keyword can be passed as argument in the constructor call.

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:


```
1.  class B{
2.    A4 obj;
3.    B(A4 obj){
4.      this.obj=obj;
5.    }
6.    void display(){
7.      System.out.println(obj.data);//using data member of A4 class
8.    }
9.  }
10.
11. class A4{
12.   int data=10;
13.   A4(){
14.    B b=new B(this);
15.    b.display();
16.   }
17.   public static void main(String args[]){
18.    A4 a=new A4();
```

```
19.  }
20. }
```

```
Output:10
```

---

6) The this keyword can be used to return current class instance.

We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1. return_type method_name(){
2. return this;
3. }
```

Example of this keyword that you return as a statement from the method

```
1. class A{
2. A getA(){
3. return this;
4. }
5. void msg(){System.out.println("Hello java");}
6. }
7.
8. class Test1{
9. public static void main(String args[]){
10. new A().getA().msg();
11. }
12. }
```

```
Output:Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
1. class A5{
2. void m(){
3. System.out.println(this);//prints same reference ID
4. }
5.
```

```
6.  public static void main(String args[]){
7.  A5 obj=new A5();
8.  System.out.println(obj);//prints the reference ID
9.
10. obj.m();
11. }
12. }
```

```
Output:A5@22b3ea59
       A5@22b3ea59
```

1. Inheritance in Java

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the IS-A relationship, also known as *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

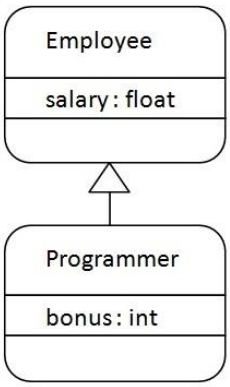
Syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.    //methods and fields
4. }
```

The extends keyword indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

Understanding the simple example of inheritance



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is Programmer IS-A Employee.It means that Programmer is a type of Employee.

```
1. class Employee{
2.  float salary=40000;
3. }
4. class Programmer extends Employee{
5.  int bonus=10000;
6.  public static void main(String args[]){
7.    Programmer p=new Programmer();
8.    System.out.println("Programmer salary is:"+p.salary);
9.    System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

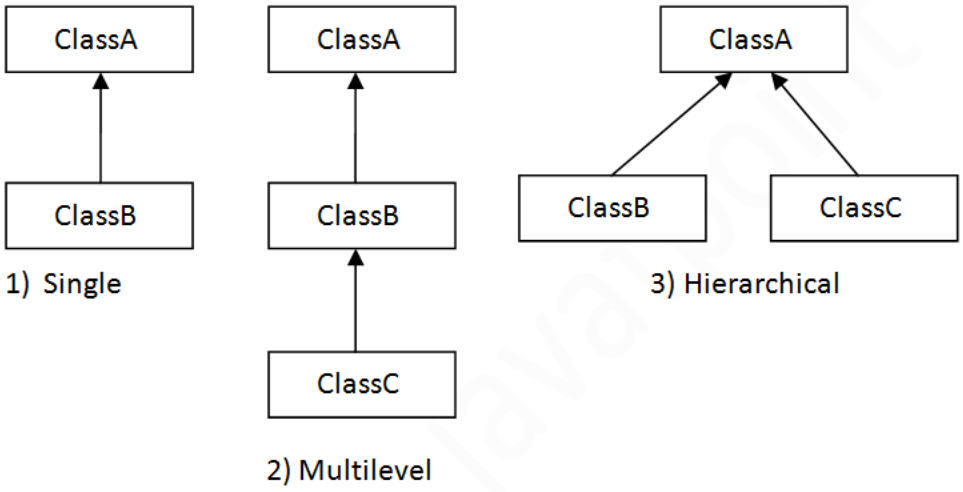
```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

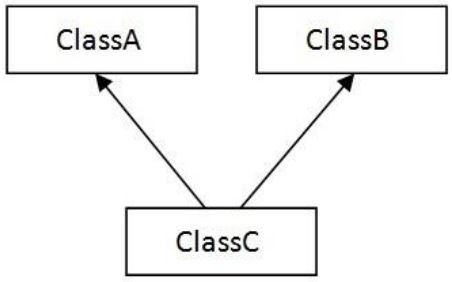
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

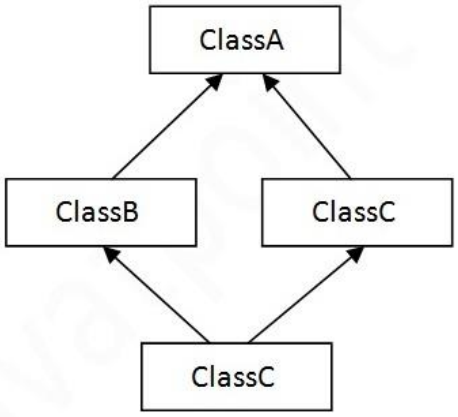


Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
1.  class A{
2.  void msg(){System.out.println("Hello");}
3.  }
4.  class B{
5.  void msg(){System.out.println("Welcome");}
6.  }
7.  class C extends A,B{//suppose if it were
8.
9.   Public Static void main(String args[]){
10.   C obj=new C();
11.   obj.msg();//Now which msg() method would be invoked?
```

12. }
13. }

```
Compile Time Error
```

2. Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

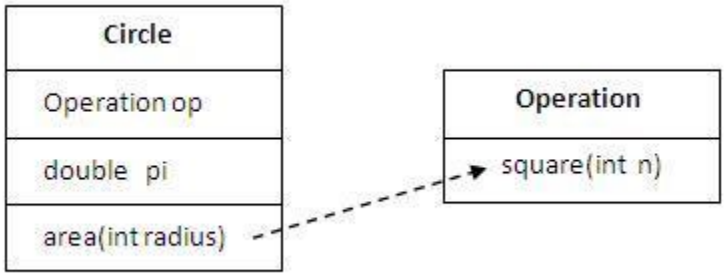
```
1. class Employee{
2. int id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
1.  class Operation{
2.   int square(int n){
3.    return n*n;
4.   }
5.  }
6.
7.  class Circle{
8.   Operation op;//aggregation
9.   double pi=3.14;
10.
11.  double area(int radius){
12.    op=new Operation();
13.    int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
14.    return pi*rsquare;
15.  }
16.
17.
18.
19.  public static void main(String args[]){
20.    Circle c=new Circle();
21.    double result=c.area(5);
22.    System.out.println(result);
23.  }
24. }
```



```
Output:78.5
```

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
1.  public class Address {
2.  String city,state,country;
3.
4.  public Address(String city, String state, String country) {
5.      this.city = city;
6.      this.state = state;
7.      this.country = country;
8.  }
9.
10. }
```

Emp.java

```
1.  public class Emp {
2.  int id;
3.  String name;
4.  Address address;
5.
6.  public Emp(int id, String name,Address address) {
7.      this.id = id;
8.      this.name = name;
9.      this.address=address;
10. }
11.
12. void display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
16.
17. public static void main(String[] args) {
18. Address address1=new Address("gzb","UP","india");
19. Address address2=new Address("gno","UP","india");
20.
21. Emp e=new Emp(111,"varun",address1);
22. Emp e2=new Emp(112,"arun",address2);
23.
24. e.display();
25. e2.display();
26.
27. }
28. }
```

```
Output:111 varun
```

```
gzb UP india
112 arun
gno UP india
```

3. Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

```
1. class A{
2. A get(){return this;}
3. }
4.
5. class B1 extends A{
6. B1 get(){return this;}
7. void message(){System.out.println("welcome to covariant return type");}
8.
9. public static void main(String args[]){
10. new B1().get().message();
11. }
12. }
```

```
Output:welcome to covariant return type
```

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type

4. super keyword in java

The super keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

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

1) super is used to refer immediate parent class instance variable.

Problem without super keyword

```
1. class Vehicle{
2.   int speed=50;
3. }
4. class Bike3 extends Vehicle{
5.   int speed=100;
6.   void display(){
7.    System.out.println(speed);//will print speed of Bike
8.   }
9.   public static void main(String args[]){
10.  Bike3 b=new Bike3();
11.  b.display();
12. }
13. }
```

```
Output:100
```

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
1.  //example of super keyword
2.
3.  class Vehicle{
4.    int speed=50;
5.  }
6.
7.  class Bike4 extends Vehicle{
8.    int speed=100;
9.
10.  void display(){
11.   System.out.println(super.speed);//will print speed of Vehicle now
12.  }
13.  public static void main(String args[]){
14.   Bike4 b=new Bike4();
15.   b.display();
16.
17. }
18. }
```

```
Output:50
```

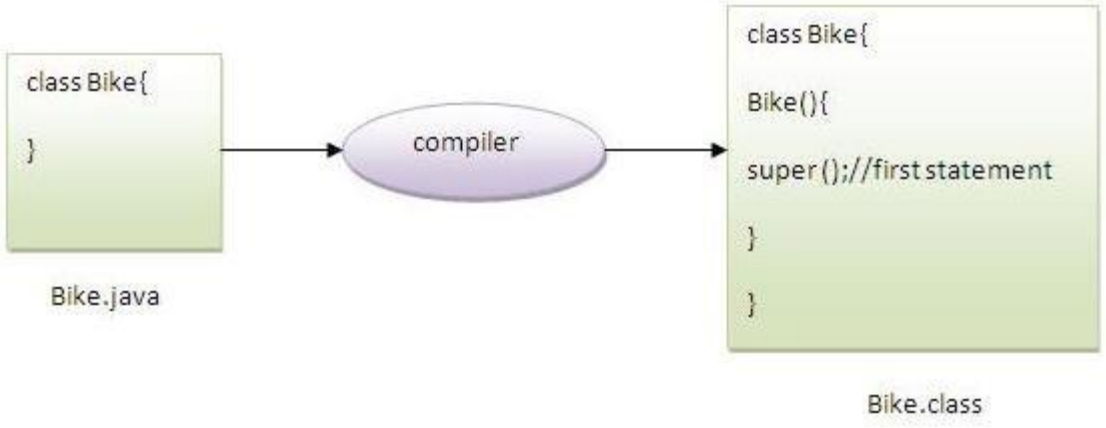

2) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor as given below:

```
1.  class Vehicle{
2.    Vehicle(){System.out.println("Vehicle is created");}
3.  }
4.
5.  class Bike5 extends Vehicle{
6.    Bike5(){
7.     super();//will invoke parent class constructor
8.     System.out.println("Bike is created");
9.    }
10.  public static void main(String args[]){
11.   Bike5 b=new Bike5();
12.
13. }
14. }
```

```
Output:Vehicle is created
       Bike is created
```

Note: super() is added in each class constructor automatically by compiler.



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

Another example of super keyword where super() is provided by the compiler implicitly.


```
1.  class Vehicle{
2.    Vehicle(){System.out.println("Vehicle is created");}
3.  }
4.
5.  class Bike6 extends Vehicle{
6.    int speed;
7.    Bike6(int speed){
8.     this.speed=speed;
9.     System.out.println(speed);
10.  }
11.  public static void main(String args[]){
12.   Bike6 b=new Bike6(10);
13. }
14. }
```

```
Output:Vehicle is created
       10
```

3) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
1. class Person{
2. void message(){System.out.println("welcome");}
3. }
4.
5. class Student16 extends Person{
6. void message(){System.out.println("welcome to java");}
7.
8. void display(){
9. message();//will invoke current class message() method
10. super.message();//will invoke parent class message() method
11. }
12.
13. public static void main(String args[]){
14. Student16 s=new Student16();
15. s.display();
16. }
17. }
```



```
Output:welcome to java
       welcome
```

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

```
1. class Person{
2. void message(){System.out.println("welcome");}
3. }
4.
5. class Student17 extends Person{
```

```
6.
7.  void display(){
8.  message();//will invoke parent class message() method
9.  }
10.
11. public static void main(String args[]){
12. Student17 s=new Student17();
13. s.display();
14. }
15. }
```

```
Output:welcome
```

5. Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. class Vehicle{
2.   void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike extends Vehicle{
5.
6.   public static void main(String args[]){
7.    Bike obj = new Bike();
8.    obj.run();
9.    }
10. }
```

```
Output:Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

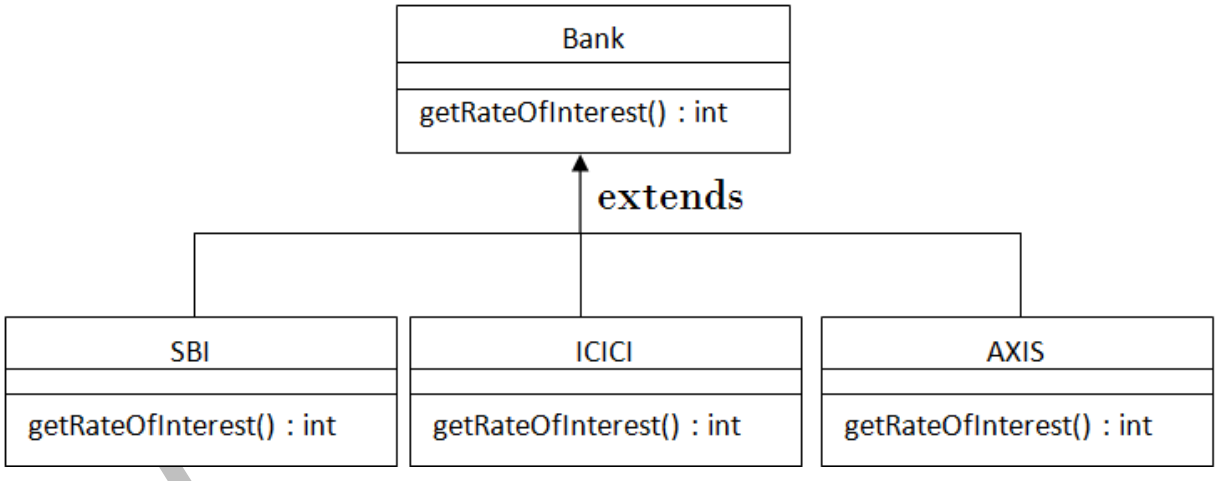
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
1. class Vehicle{
2. void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike2 extends Vehicle{
5. void run(){System.out.println("Bike is running safely");}
6.
7. public static void main(String args[]){
8. Bike2 obj = new Bike2();
9. obj.run();
10. }
```

```
Output:Bike is running safely
```

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.




```
1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
```

```
8.
9.   class ICICI extends Bank{
10. int getRateOfInterest(){return 7;}
11. }
12. class AXIS extends Bank{
13. int getRateOfInterest(){return 9;}
14. }
15.
16. class Test2{
17. public static void main(String args[]){
18. SBI s=new SBI();
19. ICICI i=new ICICI();
20. AXIS a=new AXIS();
21. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
24. }
25. }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```


Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Can we override java main method?

No, because main is a static method.

Difference between method Overloading and Method Overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

6. Instance initializer block:

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
1. class Bike{
2.     int speed=100;
3. }
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Example of instance initializer block

Let's see the simple example of instance initializer block the performs initialization.

```
1. class Bike7{
2.     int speed;
3.
4.     Bike7(){System.out.println("speed is "+speed);}
5.
6.     {speed=100;}
7.
8.     public static void main(String args[]){
9.     Bike7 b1=new Bike7();
10.    Bike7 b2=new Bike7();
11.    }
12. }
```

```
Output:speed is 100
```

```
speed is 100
```

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

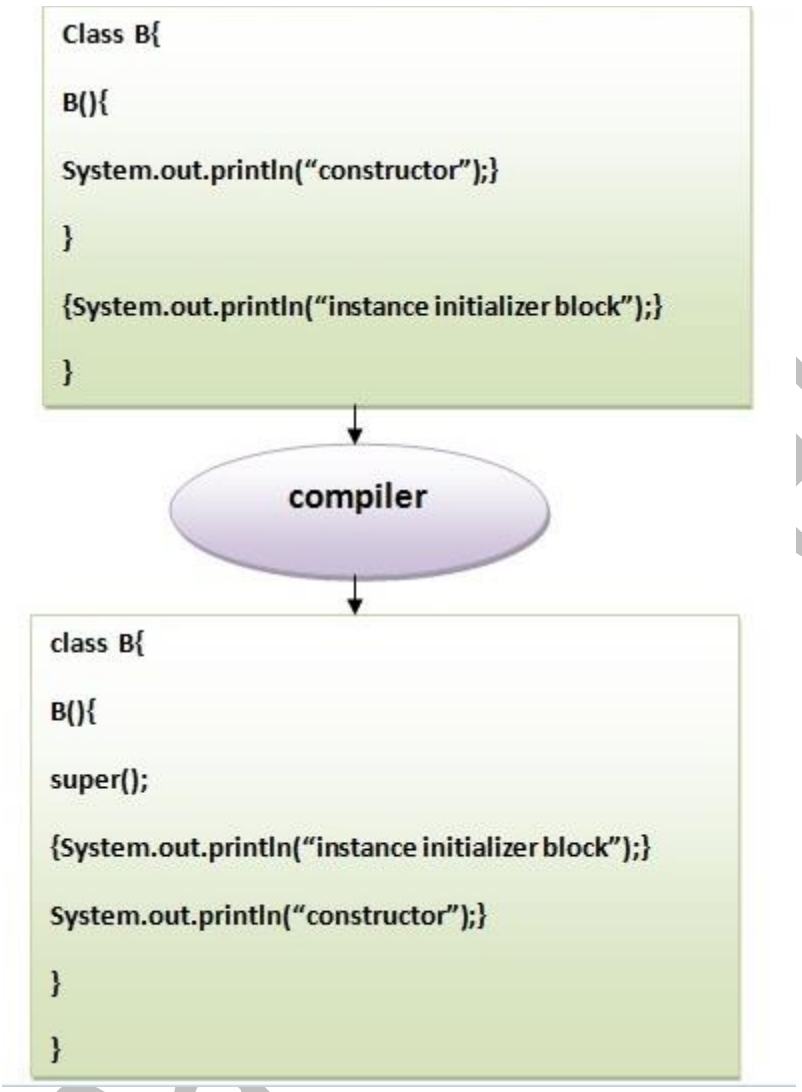
What is invoked firstly instance initializer block or constructor?

```
1.  class Bike8{
2.      int speed;
3.
4.      Bike8(){System.out.println("constructor is invoked");}
5.
6.      {System.out.println("instance initializer block invoked");}
7.
8.      public static void main(String args[]){
9.      Bike8 b1=new Bike8();
10.     Bike8 b2=new Bike8();
11.     }
12. }
```

```
Output:instance initializer block invoked
       constructor is invoked
       instance initializer block invoked
       constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.



Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
1. class A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
6. class B2 extends A{
7. B2(){
8. super();
9. System.out.println("child class constructor invoked");
10. }
11.
12. {System.out.println("instance initializer block is invoked");}
13.
14. public static void main(String args[]){
15. B2 b=new B2();
16. }
17. }
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
```


Another example of instance block

```
1. class A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
6.
7. class B3 extends A{
8. B3(){
9. super();
10. System.out.println("child class constructor invoked");
11. }
12.
13. B3(int a){
14. super();
15. System.out.println("child class constructor invoked "+a);
16. }
```

```
17.
18. {System.out.println("instance initializer block is invoked");}
19.
20. public static void main(String args[]){
21. B3 b1=new B3();
22. B3 b2=new B3(10);
23. }
24. }
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
       parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked 10
```

7. Final Keyword In Java

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

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

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1.  class Bike9{
2.   final int speedlimit=90;//final variable
3.   void run(){
4.    speedlimit=400;
5.   }
6.   public static void main(String args[]){
7.   Bike9 obj=new  Bike9();
8.   obj.run();
```

```
9.  }
10. }//end of class
```

```
Output:Compile Time Error
```

---

2) Java final method

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

Example of final method

```
1.  class Bike{
2.    final void run(){System.out.println("running");}
3.  }
4.
5.  class Honda extends Bike{
6.     void run(){System.out.println("running safely with 100kmph");}
7.
8.     public static void main(String args[]){
9.     Honda honda= new Honda();
10.    honda.run();
11.    }
12. }
```



```
Output:Compile Time Error
```

---

3) Java final class

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

Example of final class

```
1.  final class Bike{}
2.
3.  class Honda1 extends Bike{
4.    void run(){System.out.println("running safely with 100kmph");}
5.
6.    public static void main(String args[]){
7.    Honda1 honda= new Honda();
8.    honda.run();
9.    }
```

```
10. }
```

```
Output:Compile Time Error
```

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1. class Bike{
2.   final void run(){System.out.println("running...");}
3. }
4. class Honda2 extends Bike{
5.    public static void main(String args[]){
6.     new Honda2().run();
7.    }
8. }
```

```
Output:running...
```

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
1. class Student{
2. int id;
3. String name;
4. final String PAN_CARD_NUMBER;
5. ...
6. }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1.  class Bike10{
2.    final int speedlimit;//blank final variable
3.
4.    Bike10(){
5.    speedlimit=70;
6.    System.out.println(speedlimit);
7.    }
8.
9.    public static void main(String args[]){
10.     new Bike10();
11.  }
12. }
```

```
Output:70
```

---

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1.  class A{
2.    static final int data;//static blank final variable
3.    static{ data=50;}
4.    public static void main(String args[]){
5.      System.out.println(A.data);
6.   }
7.  }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1.  class Bike11{
2.    int cube(final int n){
3.     n=n+2;//can't be changed as n is final
4.     n*n*n;
```

```
5.    }
6.    public static void main(String args[]){
7.      Bike11 b=new Bike11();
8.      b.cube(5);
9.    }
10. }
```

```
Output:Compile Time Error
```

Q) Can we declare a constructor final?

No, because constructor is never inherited.

8. Polymorphism in Java

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:


```
1. class A{}
2. class B extends A{}

1. A a=new B();//upcasting
```

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

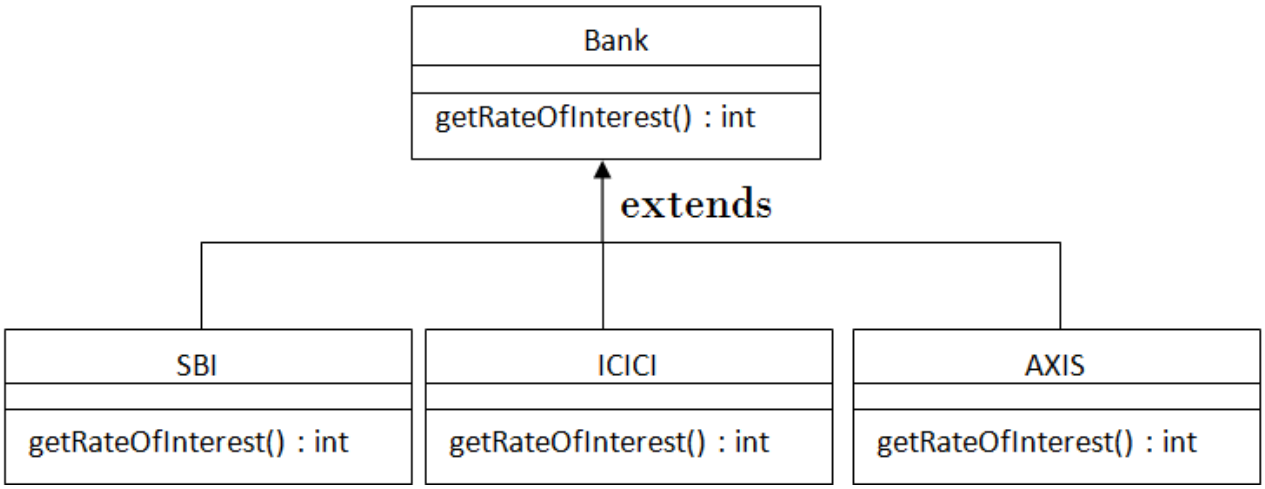
Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1. class Bike{
2.   void run(){System.out.println("running");}
3. }
4. class Splender extends Bike{
5.   void run(){System.out.println("running safely with 60km");}
6.
7.   public static void main(String args[]){
8.     Bike b = new Splender();//upcasting
9.     b.run();
10.  }
11. }
```

```
Output:running safely with 60km.
```

Real example of Java Runtime Polymorphism

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



Note: It is also given in method overriding but there was no upcasting.

```
1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
8. class ICICI extends Bank{
9. int getRateOfInterest(){return 7;}
10. }
11. class AXIS extends Bank{
12. int getRateOfInterest(){return 9;}
13. }
14. class Test3{
15. public static void main(String args[]){
16. Bank b1=new SBI();
17. Bank b2=new ICICI();
18. Bank b3=new AXIS();
19. System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
20. System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
21. System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
22. }
23. }
```



```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Java Runtime Polymorphism with data member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```
1. class Bike{
```

```
2.  int speedlimit=90;
3. }
4. class Honda3 extends Bike{
5.  int speedlimit=150;
6.
7.  public static void main(String args[]){
8.   Bike obj=new Honda3();
9.   System.out.println(obj.speedlimit);//90
10. }
```

```
Output:90
```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
1.  class Animal{
2.  void eat(){System.out.println("eating");}
3.  }
4.
5.  class Dog extends Animal{
6.  void eat(){System.out.println("eating fruits");}
7.  }
8.
9.  class BabyDog extends Dog{
10. void eat(){System.out.println("drinking milk");}
11.
12. public static void main(String args[]){
13. Animal a1,a2,a3;
14. a1=new Animal();
15. a2=new Dog();
16. a3=new BabyDog();
17.
18. a1.eat();
19. a2.eat();
20. a3.eat();
21. }
22. }
```

```
Output: eating
        eating fruits
        drinking Milk
```

Try for Output

```
1.  class Animal{
2.  void eat(){System.out.println("animal is eating...");}
3.  }
4.
5.  class Dog extends Animal{
6.  void eat(){System.out.println("dog is eating...");}
7.  }
8.
9.  class BabyDog1 extends Dog{
10. public static void main(String args[]){
11. Animal a=new BabyDog1();
12. a.eat();
13. }}
```

```
Output: Dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

9. Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
1. int data=30;
```

Here data variable is a type of int.

2) References have a type

```
1. class Dog{
2.  public static void main(String args[]){
3.   Dog d1;//Here d1 is a type of Dog
4.  }
5. }
```

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

```
1. class Animal{}
2.
3. class Dog extends Animal{
4.  public static void main(String args[]){
5.   Dog d1=new Dog();
6.  }
7. }
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

---

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
1. class Dog{
2.  private void eat(){System.out.println("dog is eating...");}
3.
4.  public static void main(String args[]){
5.   Dog d1=new Dog();
6.   d1.eat();
7.  }
8. }
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
1. class Animal{
2.  void eat(){System.out.println("animal is eating...");}
3. }
4.
5. class Dog extends Animal{
6.  void eat(){System.out.println("dog is eating...");}
7.
8.  public static void main(String args[]){
9.   Animal a=new Dog();
10.  a.eat();
11. }
12. }
```

```
Output:dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

10. Java instanceof

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
1. class Simple1{
2.  public static void main(String args[]){
3.  Simple1 s=new Simple1();
4.  System.out.println(s instanceof Simple);//true
5.  }
6. }
```

```
Output:true
```

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

```
1. class Animal{}
2. class Dog1 extends Animal{//Dog inherits Animal
3.
4.  public static void main(String args[]){
5.  Dog1 d=new Dog1();
6.  System.out.println(d instanceof Animal);//true
7.  }
8. }
```

```
Output:true
```

---

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
1. class Dog2{
```

```
2.  public static void main(String args[]){
3.   Dog2 d=null;
4.   System.out.println(d instanceof Dog2);//false
5.  }
6. }
```

```
Output:false
```

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
1. Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
1. Dog d=(Dog)new Animal();
2. //Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```
1. class Animal { }
2.
3. class Dog3 extends Animal {
4.   static void method(Animal a) {
5.     if(a instanceof Dog3){
6.       Dog3 d=(Dog3)a;//downcasting
7.       System.out.println("ok downcasting performed");
8.     }
9.   }
10.
11.  public static void main (String [] args) {
12.    Animal a=new Dog3();
13.    Dog3.method(a);
14.  }
15.
16. }
```

```
Output:ok downcasting performed
```

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
1. class Animal { }
2. class Dog4 extends Animal {
3.   static void method(Animal a) {
4.        Dog4 d=(Dog4)a;//downcasting
5.        System.out.println("ok downcasting performed");
6.   }
7.    public static void main (String [] args) {
8.     Animal a=new Dog4();
9.     Dog4.method(a);
10.  }
11. }
```

```
Output:ok downcasting performed
```

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:


```
1. Animal a=new Animal();
2. Dog.method(a);
3. //Now ClassCastException but not in case of instanceof operator
```

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```
1. interface Printable{}
2. class A implements Printable{
3. public void a(){System.out.println("a method");}
4. }
5. class B implements Printable{
6. public void b(){System.out.println("b method");}
7. }
```

```
8.
9.   class Call{
10. void invoke(Printable p){//upcasting
11. if(p instanceof A){
12. A a=(A)p;//Downcasting
13. a.a();
14. }
15. if(p instanceof B){
16. B b=(B)p;//Downcasting
17. b.b();
18. }
19.
20. }
21. }//end of Call class
22.
23. class Test4{
24. public static void main(String args[]){
25. Printable p=new B();
26. Call c=new Call();
27. c.invoke(p);
28. }
29. }
```

```
Output: b method
```

11. Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
1. abstract class A{}
```

---

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

```
1. abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4.
5. class Honda4 extends Bike{
6. void run(){System.out.println("running safely..");}
7.
8. public static void main(String args[]){
9.  Bike obj = new Honda4();
10. obj.run();
11. }
12. }
```

```
running safely..
```

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.

A factory method is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
```

```
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8.
9. class Circle1 extends Shape{
10. void draw(){System.out.println("drawing circle");}
11. }
12.
13. //In real scenario, method is called by programmer or user
14. class TestAbstraction1{
15. public static void main(String args[]){
16. Shape s=new Circle1();//In real scenario, object is provided through method e.g. getS
    hape() method
17. s.draw();
18. }
19. }
```

```
drawing circle
```

Another example of abstract class in java

File: TestBank.java

```
1.  abstract class Bank{
2. abstract int getRateOfInterest();
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 7;}
7. }
8. class PNB extends Bank{
9. int getRateOfInterest(){return 7;}
10. }
11.
12. class TestBank{
13. public static void main(String args[]){
14. Bank b=new SBI();//if object is PNB, method of PNB will be invoked
15. int interest=b.getRateOfInterest();
16. System.out.println("Rate of Interest is: "+interest+" %");
17. }}
```

```
Rate of Interest is: 7 %
```

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

```
1.  //example of abstract class that have method body
2.   abstract class Bike{
3.     Bike(){System.out.println("bike is created");}
4.     abstract void run();
5.     void changeGear(){System.out.println("gear changed");}
6.   }
7.
8.   class Honda extends Bike{
9.   void run(){System.out.println("running safely..");}
10.  }
11.  class TestAbstraction2{
12.  public static void main(String args[]){
13.   Bike obj = new Honda();
14.   obj.run();
15.   obj.changeGear();
16.  }
17. }
```

```
bike is created
running safely..
gear changed
```

Rule: If there is any abstract method in a class, that class must be abstract.

```
1.  class Bike12{
2.  abstract void run();
3.  }
```

```
compile time error
```

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1. interface A{
2. void a();
3. void b();
4. void c();
5. void d();
6. }
7.
8. abstract class B implements A{
9. public void c(){System.out.println("I am C");}
10. }
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

```
Output:I am a
       I am b
       I am c
       I am d
```

12. Access Modifiers in java

There are two types of modifiers in java: access modifiers and non-access modifiers.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

---

1) private access modifier

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
8.    A obj=new A();
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12. }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1. class A{
2. private A(){}//private constructor
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6.  public static void main(String args[]){
7.    A obj=new A();//Compile Time Error
8.  }
9. }
```

Note: A class cannot be private or protected except nested class.

---

2) default access modifier

If you don't use any modifier, it is treated as default bydefault. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1. //save by A.java
2. package pack;
3. class A{
4.   void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();//Compile Time Error
7.    obj.msg();//Compile Time Error
8.   }
9. }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

---

3) protected access modifier

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B extends A{
6.   public static void main(String args[]){
7.     B obj = new B();
8.     obj.msg();
9.   }
10. }
```

```
Output:Hello
```

---

4) public access modifier

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

```
Output:Hello
```

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2. protected void msg(){System.out.println("Hello java");}
3. }
```

```
4.
5.  public class Simple extends A{
6.  void msg(){System.out.println("Hello java");}//C.T.Error
7.   public static void main(String args[]){
8.     Simple obj=new Simple();
9.     obj.msg();
10.   }
11. }
```

The default modifier is more restrictive than protected. That is why there is compile time error.

1.Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

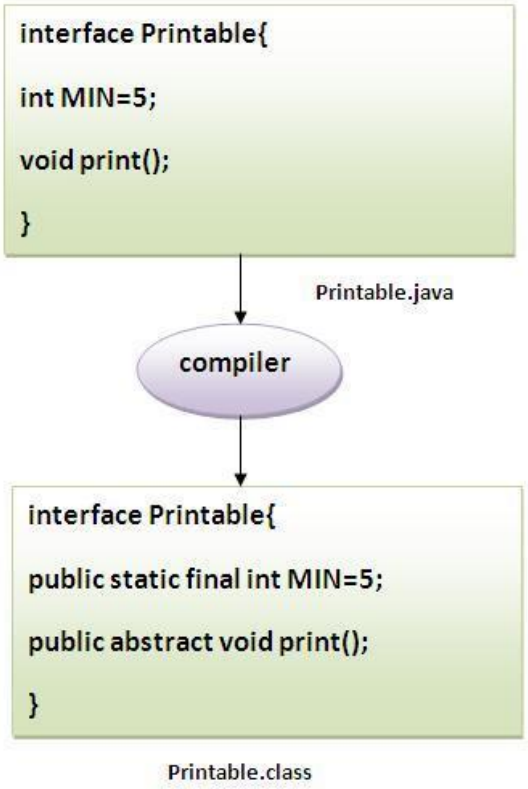
Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

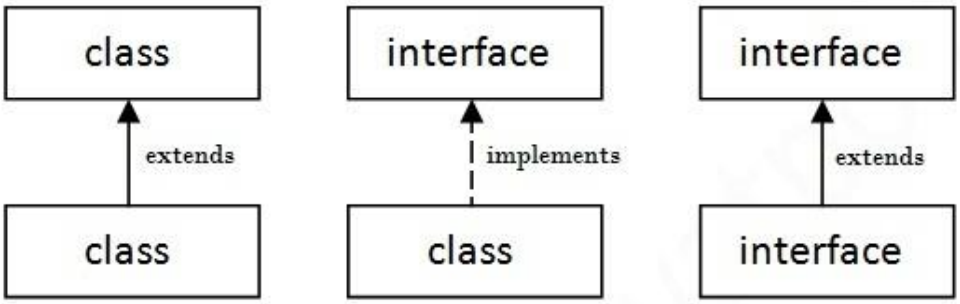
The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Simple example of Java interface

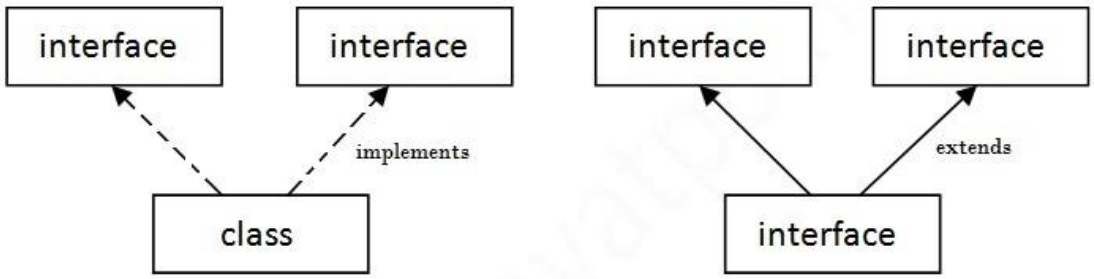
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1. interface printable{
2. void print();
3. }
4.
5. class A6 implements printable{
6. public void print(){System.out.println("Hello");}
7.
8. public static void main(String args[]){
9. A6 obj = new A6();
10. obj.print();
11. }
12. }
```

```
Output:Hello
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
1. interface Printable{
2. void print();
3. }
4.
```

```
5.  interface Showable{
6.  void show();
7.  }
8.
9.  class A7 implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12. public void show(){System.out.println("Welcome");}
13.
14. public static void main(String args[]){
15. A7 obj = new A7();
16. obj.print();
17. obj.show();
18.  }
19. }
```

```
Output:Hello
       Welcome
```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:


```
1.  interface Printable{
2.  void print();
3.  }
4.
5.  interface Showable{
6.  void print();
7.  }
8.
9.  class testinterface1 implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12.
13. public static void main(String args[]){
14. testinterface1 obj = new testinterface1();
15. obj.print();
16.  }
```

```
17. }
```

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

2.Interface inheritance

A class implements interface but one interface extends another interface .

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable extends Printable{
5.  void show();
6.  }
7.  class Testinterface2 implements Showable{
8.
9.  public void print(){System.out.println("Hello");}
10. public void show(){System.out.println("Welcome");}
11.
12. public static void main(String args[]){
13. Testinterface2 obj = new Testinterface2();
14. obj.print();
15. obj.show();
16.  }
17. }
```

```
Hello
Welcome
```

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
1.  //How Serializable interface is written?
2.  public interface Serializable{
```

```
3. }
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
1. interface printable{
2.  void print();
3.  interface MessagePrintable{
4.    void msg();
5.  }
6. }
```

3.Java Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitely.

Syntax of nested interface which is declared within the interface

```
1. interface interface_name{
2.  ...
3.  interface nested_interface_name{
4.   ...
5.  }
6. }
```


Syntax of nested interface which is declared within the class

```
1. class class_name{
2.  ...
3.  interface nested_interface_name{
4.   ...
5.  }
6. }
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
1. interface Showable{
2.   void show();
3.   interface Message{
```

```
4.     void msg();
5.    }
6.  }
7.
8.  class TestNestedInterface1 implements Showable.Message{
9.   public void msg(){System.out.println("Hello nested interface");}
10.
11.  public static void main(String args[]){
12.   Showable.Message message=new TestNestedInterface1();//upcasting here
13.   message.msg();
14.  }
15. }
```

```
Output:hello nested interface
```

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection frameword, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below:.


```
1.  public static interface Showable$Message
2.  {
3.    public abstract void msg();
4.  }
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
1.  class A{
2.    interface Message{
3.     void msg();
4.    }
5.  }
6.
7.  class TestNestedInterface2 implements A.Message{
8.   public void msg(){System.out.println("Hello nested interface");}
```

```
9.
10.  public static void main(String args[]){
11.   A.Message message=new TestNestedInterface2();//upcasting here
12.   message.msg();
13.  }
14. }
```

```
Output:hello nested interface
```

Can we define a class inside the interface?

Yes, If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
1.  interface M{
2.    class A{}
3.  }
```

4.Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.

5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1. //Creating interface that has 4 methods
2. interface A{
3. void a();//bydefault, public and abstract
4. void b();
5. void c();
6. void d();
7. }
8.
9. //Creating abstract class that provides the implementation of one method of A interface
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of t
    he methods
15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
25. a.a();
```

```
26. a.b();
27. a.c();
28. a.d();
29. }}
```

Test it Now

Output:

```
I am a
I am b
I am c
I am d
```

1 Exception Handling in Java

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

1.1 What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

1.2 What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

1.3 Advantage of Exception Handling

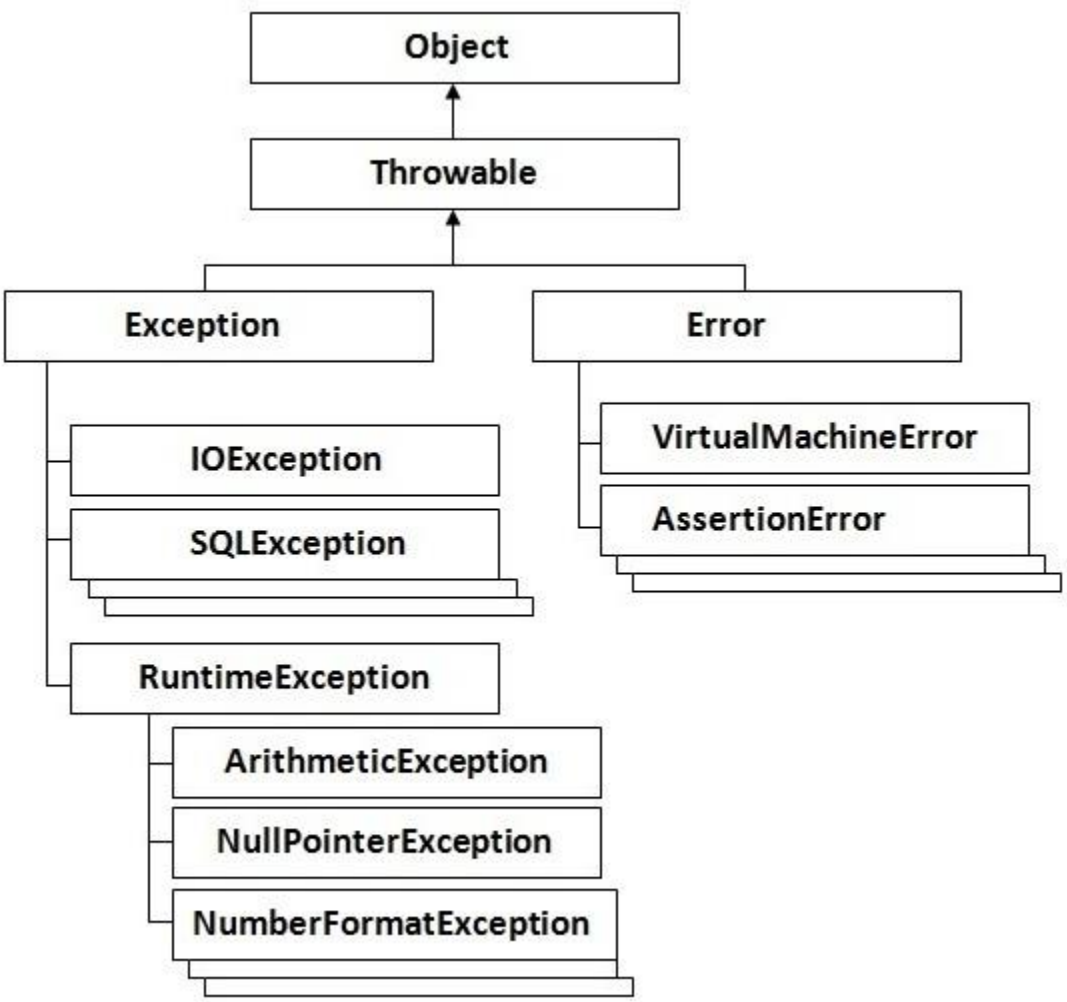
The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform

exception handling, rest of the exception will be executed. That is why we use exception handling in java.

Hierarchy of Java Exception classes



1.4 Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

---

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

---

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

---

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

1.5 Java try-catch

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

```
1. try{
2. //code that may throw exception
3. }catch(Exception_class_Name ref){}
```

1.5.1.1 Syntax of try-finally block

```
1. try{
2. //code that may throw exception
3. }finally{}
```

1.6 Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
1. public class Testtrycatch1{
2.   public static void main(String args[]){
3.     int data=50/0;//may throw exception
4.     System.out.println("rest of the code...");
5. }
6. }
```

Output:

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

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

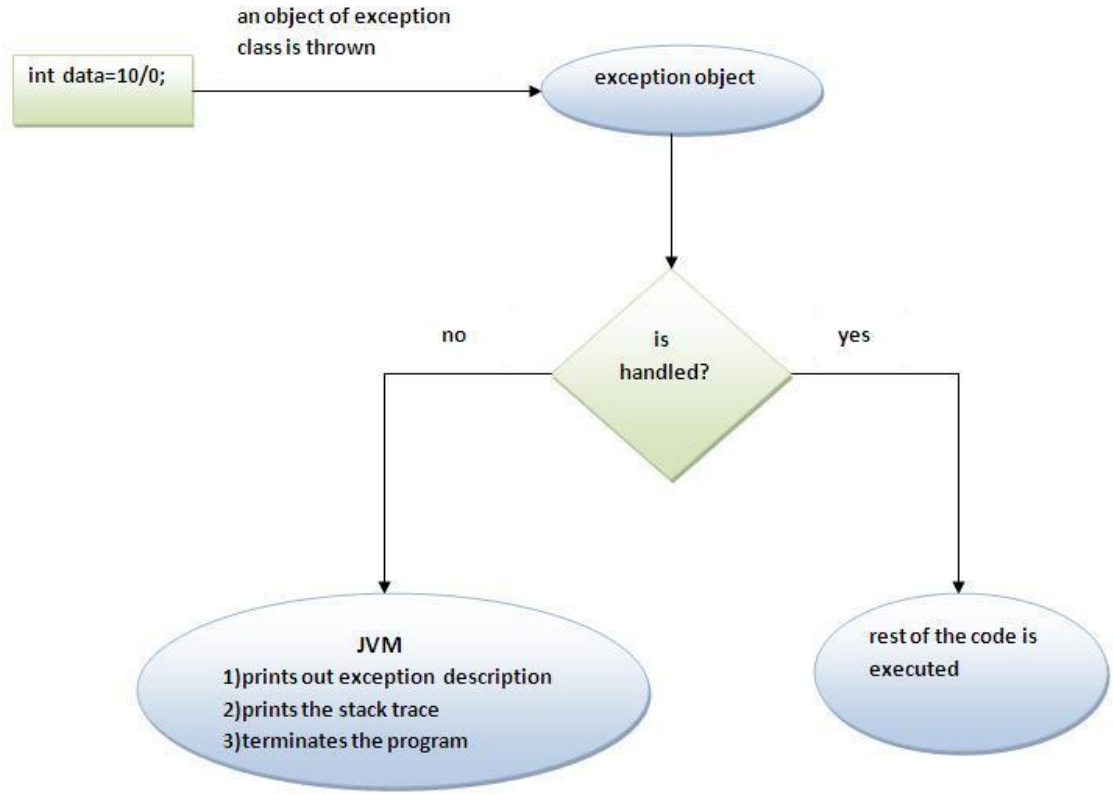
```
1.  public class Testtrycatch2{
2.    public static void main(String args[]){
3.     try{
4.        int data=50/0;
5.     }catch(ArithmeticException e){System.out.println(e);}
6.     System.out.println("rest of the code...");
7.  }
8.  }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

1.7 Java catch multiple exceptions

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1. public class TestMultipleCatchBlock{
2.   public static void main(String args[]){
3.    try{
```

```
4.      int a[]=new int[5];
5.      a[5]=30/0;
6.     }
7.     catch(ArithmeticException e){System.out.println("task1 is completed");}
8.     catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9.     catch(Exception e){System.out.println("common task completed");}
10.
11.    System.out.println("rest of the code...");
12.  }
13. }
```

```
Output:task1 completed
       rest of the code...
```

- **Rule: At a time only one Exception is occured and at a time only one catch block is executed.**

- **Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .**

```
1.  class TestMultipleCatchBlock1{
2.    public static void main(String args[]){
3.     try{
4.      int a[]=new int[5];
5.      a[5]=30/0;
6.     }
7.     catch(Exception e){System.out.println("common task completed");}
8.     catch(ArithmeticException e){System.out.println("task1 is completed");}
9.     catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10.    System.out.println("rest of the code...");
11.  }
12. }
```

Output:

```
Compile-time error
```

1.8 Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
1. ....
2. try
3. {
4.     statement 1;
5.     statement 2;
6.     try
7.     {
8.         statement 1;
9.         statement 2;
10.    }
11.    catch(Exception e)
12.    {
13.    }
14. }
15. catch(Exception e)
16. {
17. }
18. ....
```

Java nested try example

Let's see a simple example of java nested try block.

```
1.  class Excep6{
2.   public static void main(String args[]){
3.    try{
4.     try{
5.      System.out.println("going to divide");
6.      int b =39/0;
7.     }catch(ArithmeticException e){System.out.println(e);}
8.
9.     try{
10.    int a[]=new int[5];
11.    a[5]=4;
```

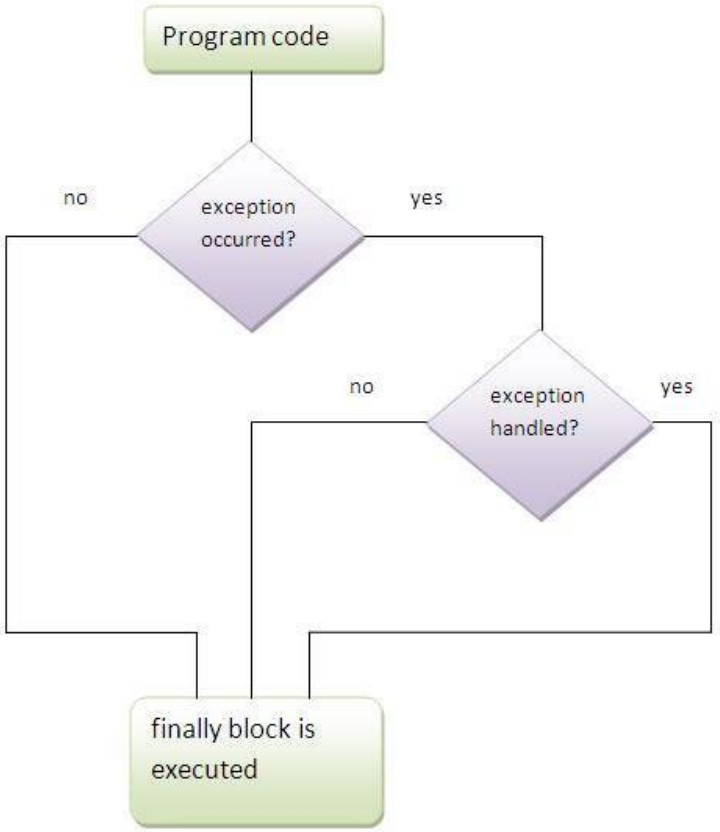
```
12.     }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.     System.out.println("other statement);
15.    }catch(Exception e){System.out.println("handeled");}
16.
17.    System.out.println("normal flow..");
18.   }
19. }
```

1.9 finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1.  class TestFinallyBlock{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/5;
5.     System.out.println(data);
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11. }
```

```
Output:5
       finally block is always executed
       rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1.  class TestFinallyBlock1{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
```

```
10.  }
11. }
```

```
Output:finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1.  public class TestFinallyBlock2{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
6.    }
7.    catch(ArithmeticException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.  }
11. }
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```


- **Rule: For each try block there can be zero or more catch blocks, but only one finally block.**
- **Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).**

1.10 Java throw exception

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

1. throw exception;

Let's see the example of throw IOException.

1. throw new IOException("sorry device error);

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1.  public class TestThrow1{
2.     static void validate(int age){
3.       if(age<18)
4.        throw new ArithmeticException("not valid");
5.       else
6.        System.out.println("welcome to vote");
7.     }
8.     public static void main(String args[]){
9.       validate(13);
10.      System.out.println("rest of the code...");
11.  }
12. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

1.11 Java Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous

method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

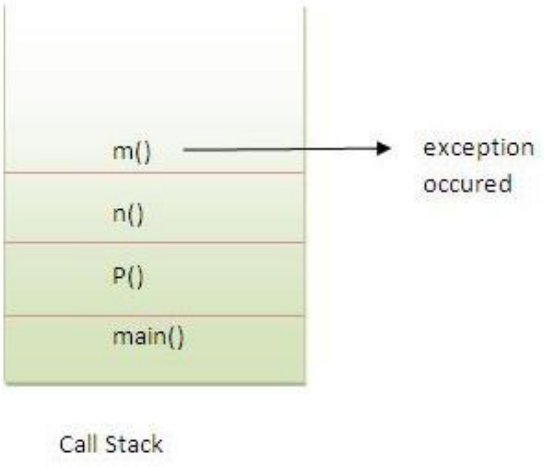
1.11.1.1 Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

```
1.  class TestExceptionPropagation1{
2.    void m(){
3.     int data=50/0;
4.    }
5.    void n(){
6.     m();
7.    }
8.    void p(){
9.     try{
10.    n();
11.   }catch(Exception e){System.out.println("exception handled");}
12.  }
13.  public static void main(String args[]){
14.   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
15.   obj.p();
16.   System.out.println("normal flow...");
17.  }
18. }
```

Test it Now

```
Output:exception handled
       normal flow...
```



Call Stack

In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

1.11.1.2 Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
1.  class TestExceptionPropagation2{
2.    void m(){
3.      throw new java.io.IOException("device error");//checked exception
4.    }
5.    void n(){
6.      m();
7.    }
8.    void p(){
9.     try{
10.     n();
11.    }catch(Exception e){System.out.println("exception handeled");}
12.   }
13.   public static void main(String args[]){
14.    TestExceptionPropagation2 obj=new TestExceptionPropagation2();
15.    obj.p();
16.    System.out.println("normal flow");
17.   }
18. }
```

```
Output:Compile Time Error
```

1.12 Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1. import java.io.IOException;
2. class Testthrows1{
3.   void m()throws IOException{
4.     throw new IOException("device error");//checked exception
5.   }
6.   void n()throws IOException{
7.     m();
8.   }
9.   void p(){
10.   try{
11.    n();
12.   }catch(Exception e){System.out.println("exception handled");}
13.  }
14.  public static void main(String args[]){
15.   Testthrows1 obj=new Testthrows1();
16.   obj.p();
```

```
17.    System.out.println("normal flow...");
18.   }
19. }
```

Output:

```
exception handled
normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    throw new IOException("device error");
5.   }
6.  }
7.  public class Testthrows2{
8.     public static void main(String args[]){
9.      try{
10.      M m=new M();
11.      m.method();
12.     }catch(Exception e){System.out.println("exception handled");}
13.
14.     System.out.println("normal flow...");
15.   }
16. }
```

```
Output:exception handled
       normal flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   System.out.println("device operation performed");
5.  }
6. }
7. class Testthrows3{
8.    public static void main(String args[])throws IOException{//declare exception
9.      M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14. }
```

```
Output:device operation performed
       normal flow...
```

B)Program if exception occurs

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   throw new IOException("device error");
5.  }
6. }
7. class Testthrows4{
8.    public static void main(String args[])throws IOException{//declare exception
9.      M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14. }
```

```
Output:Runtime Exception
```

1.13 Difference between throw and throws

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

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

Java throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
3. }
```

1.14 Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1. class FinalExample{
2. public static void main(String[] args){
3. final int x=100;
4. x=200;//Compile Time Error
5. }}
```

Java finally example

```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{System.out.println("finally block is executed");}
7. }}
```

Java finalize example

```
1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc();
9. }}
```

1.15 ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class TestExceptionChild extends Parent{
7.   void msg()throws IOException{
8.     System.out.println("TestExceptionChild");
9.   }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild();
12.   p.msg();
13.  }
14. }
```

```
Output:Compile Time Error
```

---

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
```

```
5.
6.  class TestExceptionChild1 extends Parent{
7.    void msg()throws ArithmeticException{
8.      System.out.println("child");
9.    }
10.   public static void main(String args[]){
11.    Parent p=new TestExceptionChild1();
12.    p.msg();
13.   }
14. }
```

```
Output:child
```

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

1.15.1 Example in case subclass overridden method declares parent exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws ArithmeticException{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild2 extends Parent{
7.    void msg()throws Exception{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.    Parent p=new TestExceptionChild2();
11.    try{
12.     p.msg();
13.    }catch(Exception e){}
14.   }
15. }
```

```
Output:Compile Time Error
```

1.15.2 Example in case subclass overridden method declares same exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws Exception{System.out.println("parent");}
```

```
4.  }
5.
6.  class TestExceptionChild3 extends Parent{
7.    void msg()throws Exception{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.   Parent p=new TestExceptionChild3();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15. }
```

```
Output:child
```

1.15.3 Example in case subclass overridden method declares subclass exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws Exception{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild4 extends Parent{
7.    void msg()throws ArithmeticException{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.   Parent p=new TestExceptionChild4();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15. }
```

```
Output:child
```

1.15.4 Example in case subclass overridden method declares no exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws Exception{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild5 extends Parent{
```

```
7.    void msg(){System.out.println("child");}
8.
9.    public static void main(String args[]){
10.    Parent p=new TestExceptionChild5();
11.    try{
12.    p.msg();
13.    }catch(Exception e){}
14.   }
15. }
```

1.16 Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
1.  class InvalidAgeException extends Exception{
2.   InvalidAgeException(String s){
3.    super(s);
4.   }
5.  }
```



```
1.  class TestCustomException1{
2.
3.     static void validate(int age)throws InvalidAgeException{
4.       if(age<18)
5.        throw new InvalidAgeException("not valid");
6.       else
7.        System.out.println("welcome to vote");
8.     }
9.
10.    public static void main(String args[]){
11.       try{
12.       validate(13);
13.       }catch(Exception m){System.out.println("Exception occured: "+m);}
14.
15.       System.out.println("rest of the code...");
16.   }
17. }
```

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```