**Java’s Lineage**

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java’s object-oriented features were influenced by C++. In fact, several of Java’s defining characteristics come from—or are responses to—its predecessors. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is the outcome of the same terminology.

which was initiated by James Gosling and released in 1995developed by [Sun Microsystems](https://en.wikipedia.org/wiki/Sun_Microsystems) as Java 1.0,

Java programming language was originally developed by Sun Microsystems

which was later acquired by the [Oracle Corporation](https://en.wikipedia.org/wiki/Oracle_Corporation),

On November 13, 2006, Sun Microsystems made the bulk of its implementation of Java available under the [GNU General Public License](https://en.wikipedia.org/wiki/GNU_General_Public_License) (GPL).

As of December 2008, the latest release of the Java Standard Edition is 6 (J2SE).

The latest version only supported version as of 2016.

that provides a system for developing [application software](https://en.wikipedia.org/wiki/Application_software) and deploying it in a [cross-platform](https://en.wikipedia.org/wiki/Cross-platform) computing environment. Java is used in a wide variety of [computing platforms](https://en.wikipedia.org/wiki/Computing_platform) from [embedded devices](https://en.wikipedia.org/wiki/Embedded_device) and [mobile phones](https://en.wikipedia.org/wiki/Mobile_phone) to [enterprise servers](https://en.wikipedia.org/wiki/Enterprise_server) and [supercomputers](https://en.wikipedia.org/wiki/Supercomputer). While they are less common than standalone Java applications, [Java applets](https://en.wikipedia.org/wiki/Java_applet) run in secure, [sandboxed](https://en.wikipedia.org/wiki/Sandbox_%28computer_security%29) environments to provide many features of native applications and can be embedded in [HTML](https://en.wikipedia.org/wiki/HTML) pages

is Java 8, the

**Java’s Past, Present, and Future**

The Java language was developed at Sun Microsystems in 1991 as part of a research project to

develop software for consumer electronics devices—television sets, VCRs, toasters, and the

other sorts of machines you can buy at any department store. Java’s goals at that time were to

be small, fast, efficient, and easily portable to a wide range of hardware devices. It is those same

goals that made Java an ideal language for distributing executable programs via the World Wide

Web, and also a general-purpose programming language for developing programs that are easily

usable and portable across different platforms.

The Java language was used in several projects within Sun, but did not get very much commercial

attention until it was paired with HotJava. HotJava was written in 1994 in a matter of months,

both as a vehicle for downloading and running applets and also as an example of the sort of

complex application that can be written in Java.

At the time this book is being written, Sun has released the beta version of the Java Developer’s

Kit (JDK), which includes tools for developing Java applets and applications on Sun systems

running Solaris 2.3 or higher for Windows NT and for Windows 95. By the time you read this,

support for Java development may have appeared on other platforms, either from Sun or from

third-party companies.

Note that because the JDK is currently in beta, it is still subject to change between now and when

it is officially released. Applets and applications you write using the JDK and using the examples

in this book may require some changes to work with future versions of the JDK. However,

because the Java language has been around for several years and has been used for several projects,

the language itself is quite stable and robust and most likely will not change excessively. Keep

this beta status in mind as you read through this book and as you develop your own Java

programs.

Support for playing Java programs is a little more confusing at the moment. Sun’s HotJava is

not currently included with the Beta JDK; the only available version of HotJava is an older alpha

version, and, tragically, applets written for the alpha version of Java do not work with the beta

JDK, and vice versa. By the time you read this, Sun may have released a newer version of HotJava

which will enable you to view applets.

The JDK does include an application called appletviewer that allows you to test your Java applets

as you write them. If an applet works in the appletviewer, it should work with any Java-capable

browser. You’ll learn more about applet viewer later today.

What’s in store for the future? In addition to the final Java release from Sun, other companies

have announced support for Java in their own World Wide Web browsers. Netscape Communications

Corporation has already incorporated Java capabilities into the 2.0 version of their very

popular Netscape Navigator Web browser—pages with embedded Java applets can be viewed

and played with Netscape. With support for Java available in as popular a browser as Netscape, tools to help develop Java applications (debuggers, development environments, and so on) most

likely will be rapidly available as well.

There are 3 levels of Java.

a. J2SE - Java 2 Standard Edition (Core Java)

b. J2EE - Java 2 Enterprise Edition (Adv. Ent. Java)

b. J2ME - Java 2 Micro Edition (java for mobile and hand held devices)

**Java characteristics & features**

**11 Features of Java Programming Language**

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

**Simple**

|  |
| --- |
| According to Sun, Java language is simple because: |
| * Most of the concepts and Syntax is based on C++ thus simple for programmers to learn it after C++. * Java is Easy to write and more readable and eye catching. * It removed many confusing and / or rarely used features e.g., explicit pointers, operator overloading, etc. |
| * Java has a concise, cohesive set of features that makes it easy to learn and use. |
| * No need to remove unreferenced objects because there is Automatic Garbage Collection in java. |

**Object-oriented**

|  |
| --- |
| * Java programming is object-oriented programming language. * Like C++ java provides most of the object oriented features. * Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. |
| * Object-oriented programming (OOPs) is a methodology that simplify software development and maintenance by providing some rules. * Java is pure OOP. Language. (while C++ is semi object oriented) |
| Basic concepts of OOPs are: |
| 1. Object 2. Class 3. Inheritance 4. Polymorphism 5. Abstraction 6. Encapsulation |

**Platform Independent**

|  |
| --- |
| A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms. It has two components:   1. Runtime Environment 2. API(Application Programming Interface) |

|  |
| --- |
| java is platform independent  Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms i.e. the main concept is “Write Once and Run Any Where (WORA)”. |

**Secured**

|  |
| --- |
| Java is secured because: |
| * No explicit pointer * Programs run inside virtual machine sandbox. |

|  |  |
| --- | --- |
| how java is secured | how java is secured |

|  |
| --- |
| * **Classloader-** adds security by separating the package for the classes of the local file system from those that are imported from network sources. * **Bytecode Verifier-** checks the code fragments for illegal code that can violate access right to objects. * **Security Manager-** determines what resources a class can access such as reading and writing to the local disk. |
| These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, cryptography etc. |

**Robust**

|  |
| --- |
| Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust. |

**Architecture-neutral**

|  |
| --- |
| There is no implementation dependent features e.g. size of primitive types is set. |

**Portable**

|  |
| --- |
| We may carry the java bytecode to any platform. |

**High-performance**

|  |
| --- |
| Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++) |

**Distributed**

|  |
| --- |
| We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet. |

**Multi-threaded**

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

## JDK (Java Development Kit)

JDK contains everything that will be required to ***develop and run*** Java application.

## JRE (Java Run time Environment)

JRE contains everything required to ***run*** Java application which has already been compiled. It doesn’t contain the code library required to develop Java application.

## JVM (Java Virtual Machine)

JVM is a virtual machine which work on top of your operating system to provide a recommended environment for your compiled Java code. JVM only works with bytecode. Hence you need to compile your Java application(.java) so that it can be converted to bytecode format (also known as the .class file). Which then will be used by JVM to run application. JVM only provide the environment It needs the Java code library to run applications.

Data Types or **Types :**  
  
The Java programming language has many built in data types. These are classified into two broad categories

1. Primitive Types
2. Reference Data Type

Primitive types are simple values and not objects. reference types are used for more complex types, includeing all types that we declare ourselves.

**The Primitive Types**

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and

**boolean**. The primitive types are also commonly referred to as *simple* types. These can be put in four groups:

**• Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers. The most commonly used integer type is **int**. Variables of type **int** are often employed to

control loops, to index arrays, and to perform general-purpose integer math.

**• Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.

There are two kinds of floating-point types, **float** and **double**, which represent

single- and double-precision numbers, respectively. Type **float** is 32 bits wide and type **double**

is 64 bits wide.

Ex.

// Compute the area of a circle.

class Area {

public static void main(String args[]) {

double pi, r, a;

r = 10.8; // radius of circle

pi = 3.1416; // pi, approximately

a = pi \* r \* r; // compute area

System.out.println("Area of circle is " + a);

}

}

**• Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers. In Java, characters are not 8-bit quantities like they are in most other computer languages.

A character variable can be assigned a value by enclosing the character in single quotes.

For example, this assigns the variable **ch** the letter X:

char ch;

ch = 'X';

Ex.

// Demonstrate char data type.

class CharDemo {

public static void main(String args[]) {

char ch1, ch2;

ch1 = 88; // code for X

ch2 = 'Y';

System.out.print("ch1 and ch2: ");

System.out.println(ch1 + " " + ch2);

}

}

Output:

ch1 and ch2: X Y

**• Boolean** This group includes **boolean**, which is a special type for representing true/false values.

Java defines the values true and false using the

reserved words **true** and **false**.

Ex.

// Demonstrate boolean values.

class BoolTest {

public static void main(String args[]) {

boolean b;

b = false;

System.out.println("b is " + b);

b = true;

System.out.println("b is " + b);

// a boolean value can control the if statement

if(b) System.out.println("This is executed.");

b = false;

if(b) System.out.println("This is not executed.");

// outcome of a relational operator is a boolean value

System.out.println("10 > 9 is " + (10 > 9));

}

}

Output:

b is false

b is true

This is executed.

10 > 9 is true

Reference Data Types:

Reference variables are created using defined constructors of the classes. They are used to access objects.

These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy,

etc.

Class objects and various types of array variables come under reference data type.

Default value of any reference variable is null.

A reference variable can be used to refer to any object of the declared type or any compatible type.

Example: Animal animal = new Animal("giraffe");

**Identifiers:**

All components in the Java program require names as their identity. The main components are classes, variables and methods and the Names used to identify them are called identifiers.

**Rules for using Java Identifiers:**

All identifiers should begin with alphabets (both upper case and lower case are allowed e.g. A to Z or a to z) or special characters i.e. currency character ($) or an underscore (\_).

After the first character, identifiers can have any combination of characters.

A keyword cannot be used as an identifier.

Most importantly identifiers are case sensitive.

Example: Valid identifiers: age, $salary, \_value

Invalid identifiers: 123abc, -salary

**Escape Sequences**

A character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler. When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

The following table shows the Java escape sequences:

|  |  |
| --- | --- |
| **Escape Sequences** | |
| **Escape Sequence** | **Description** |
| \t | Insert a tab in the text at this point. |
| \b | Insert a backspace in the text at this point. |
| \n | Insert a newline in the text at this point. |
| \r | Insert a carriage return in the text at this point. |
| \f | Insert a formfeed in the text at this point. |
| \' | Insert a single quote character in the text at this point. |
| \" | Insert a double quote character in the text at this point. |
| \\ | Insert a backslash character in the text at this point. |

Input

//beginners guide

There is no direct parallel to the very convenient

**println( )** method, for example, that allows you to read various types of data entered by the

user. Frankly, Java’s approach to console input is not as easy to use as one might like. Also,

most real-world Java programs and applets will be graphical and window based, not console

based. For these reasons, not much use of console input is found in this book. However, there

is one type of console input that *is* easy to use: reading a character from the keyboard. Since

several of the examples in this module will make use of this feature, it is discussed here.

The easiest way to read a character from the keyboard is to call **System.in.read( )**.

**System.in** is the complement to **System.out**. It is the input object attached to the keyboard.

The **read( )** method waits until the user presses a key and then returns the result. The character

is returned as an integer, so it must be cast into a **char** to assign it to a **char** variable. By default,

console input is line buffered, so you must press ENTER before any character that you type will

be sent to your program. Here is a program that reads a character from the keyboard:

**Java I/O system**

Java I/O system is based upon a hierarchy of classes, it was not possible to present its theory and details without first discussing classes, inheritance, and exceptions.

Java’s I/O system is quite large, containing many classes, interfaces, and methods.

The reason for its size is that Java defines two complete I/O systems: one for byte I/O and the other for character I/O.

Java’s I/O system is cohesive and consistent; once you understand its fundamentals, the rest of the I/O system is easy to master.

most real applications of Java will not be text-based, console programs. Rather, they will be graphically oriented

programs, such as applets, that rely upon a windowed interface for interaction with the user.

* Java’s I/O Is Built upon Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces

or consumes information. With stream the same I/O classes and methods can be applied to any type of device. For

example, the same methods that you use to write to the console can also be used to write to

a disk file. Java implements streams within class hierarchies defined in the **java.io** package.

Byte Streams and Character Streams

Modern versions of Java define two types of streams: byte and character. (The original version

of Java defined only the byte stream, but character streams were quickly added.)

Byte streams

provide a convenient means for handling input and output of bytes. They are used, for example,

when reading or writing binary data. They are especially helpful when working with files.

Character streams are designed for handling the input and output of characters. They use

Unicode and, therefore, can be internationalized. Also, in some cases, character streams are

more efficient than byte streams.

**Note:** At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

Naming conventions in java

## Naming conventions in java

|  |  |
| --- | --- |
| **Name** | **Convention** |
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX\_PRIORITY etc. |
|  |  |

**Keywords / Reserve Words in JAVA**

There are some words that you cannot use as object or variable names in a Java program. These words are known as “reserved” words; they are keywords that are already used by the syntax of the [Java programming](http://java.about.com/od/gettingstarted/fl/How-Do-I-Start-Learning-Java.htm) language.

**The table below lists all the words that are reserved:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| abstract | assert | boolean | break | byte | case |
| catch | char | class | const\* | continue | default |
| double | do | else | enum | extends | false |
| final | finally | float | for | goto\* | if |
| implements | import | instanceof | int | interface | long |
| native | new | null | package | private | protected |
| public | return | short | static | strictfp | super |
| switch | synchronized | this | throw | throws | transient |
| true | try | void | volatile | while |  |

For example, if you try and create a [new class](http://java.about.com/od/c/g/Class.htm) and name it using a [reserved word](http://javascript.about.com/od/reference/g/greserved.htm):

// you can't use finally as it's a reserved word!

class finally {

   public static void main(String[] args) {

      //class code..

   }

}

It will not compile, instead you will get the following error:

<identifier> expected

**Variables**

The variable in JAVA program work as the basic unit of storage. A variable is defined by the combination of an identifier, a type, a scope and an optional initializer. The scope of the variable defines their visibility, and a lifetime.

In Java, all variables must be declared before they can be used.

Syntax for declaring variables is:

*datatype identifier* [ = *value* ][, *identifier* [= *value* ] …];

Here, *type* is one of Java’s atomic types, or the name of a class or interface. (Class and

interface types are discussed later in Part I of this book.) The *identifier* is the name of the

variable. You can initialize the variable by specifying an equal sign and a value. Keep in

mind that the initialization expression must result in a value of the same (or compatible)

type as that specified for the variable. To declare more than one variable of the specified type,

use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some

include an initialization.

int a, b, c; // declares three ints, a, b, and c.

int d = 3, e, f = 5; // declares three more ints, initializing

// d and f.

byte z = 22; // initializes z.

double pi = 3.14159; // declares an approximation of pi.

char x = 'x'; // the variable x has the value 'x'.

The identifiers that you choose have nothing intrinsic in their names that indicates

their type. Java allows any properly formed identifier to have any declared type.

**Types and Scope of the Variable**

There are three kinds of variables in Java:

• Local variables

• Instance variables

• Class/static variables

Local variables:

• Local variables are declared in methods, constructors, or blocks.

• Local variables are created when the method, constructor or block is entered and the variable will be destroyed

once it exits the method, constructor or block.

• Access modifiers cannot be used for local variables.

Local variables are visible only within the declared method, constructor or block.

• Local variables are implemented at stack level internally.

• There is no default value for local variables so local variables should be declared and an initial value should be

assigned before the first use.

Example:

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

public class Test{

public void pupAge(){

int age = 0;

age = age + 7;

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

}

public static void main(String args[]){

Test test = new Test();

test.pupAge();

}

}

This would produce the following result:

Puppy age is: 7

Example:

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

public class Test{

public void pupAge(){

int age;

age = age + 7;

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

}

public static void main(String args[]){

Test test = new Test();

test.pupAge();

}

}

This would produce the following error while compiling it:

Test.java:4:variable number might not have been initialized

age = age + 7;

^

1 error

Instance

variables:

• Instance variables are declared in a class, but outside a method, constructor or any block.

When a space is allocated for an object in the heap, a slot for each instance variable value is created.

• Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when

the object is destroyed.

• Instance variables hold values that must be referenced by more than one method, constructor or block, or

essential parts of an object's state that must be present throughout the class.

• Instance variables can be declared in class level before or after use.

• Access modifiers can be given for instance variables.

• The instance variables are visible for all methods, constructors and block in the class. Normally, it is

recommended to make these variables private (access level). However visibility for subclasses can be given for

these variables with the use of access modifiers.

• Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object

references it is null. Values can be assigned during the declaration or within the constructor.

• Instance variables can be accessed directly by calling the variable name inside the class. However within static

methods and different class ( when instance variables are given accessibility) should be called using the fully

qualified name . *ObjectReference.VariableName*.

Example:

import java.io.\*;

public class Employee{

// this instance variable is visible for any child class.

public String name;

// salary variable is visible in Employee class only.

private double salary;

// The name variable is assigned in the constructor.

public Employee (String empName){

name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal){

salary = empSal;

}

// This method prints the employee details.

public void printEmp(){

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

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

}

public static void main(String args[]){

Employee empOne = new Employee("Ransika");

empOne.setSalary(1000);

empOne.printEmp();

}

}

This would produce the following result:

name : Ransika

salary :1000.0

**Class/static variables:**

• Class variables also known as static variables are declared with the *static* keyword in a class, but outside a

method, constructor or a block.

• There would only be one copy of each class variable per class, regardless of how many objects are created

from it.

• Static variables are rarely used other than being declared as constants. Constants are variables that are

declared as public/private, final and static. Constant variables never change from their initial value.

• Static variables are stored in static memory. It is rare to use static variables other than declared final and used

as either public or private constants.

• Static variables are created when the program starts and destroyed when the program stops.

• Visibility is similar to instance variables. However, most static variables are declared public since they must be

available for users of the class.

• Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and

for object references, it is null. Values can be assigned during the declaration or within the constructor.

Additionally values can be assigned in special static initializer blocks.

• Static variables can be accessed by calling with the class name . *ClassName.VariableName*.

• When declaring class variables as public static final, then variables names (constants) are all in upper case. If

the static variables are not public and final the naming syntax is the same as instance and local variables.

**Example:**

import java.io.\*;

public class Employee{

// salary variable is a private static variable

private static double salary;

// DEPARTMENT is a constant

public static final String DEPARTMENT = "Development ";

public static void main(String args[]){

salary = 1000;

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

}

}

This would produce the following result:

Development average salary:1000

**Operators**

Java provides a rich operator environment. These operators are used to manipulate variables. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

Arithmetic Operators

Relational Operators

Bitwise Operators

Logical Operators

Assignment Operators

**Arithmetic Operators**

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The

following table lists the arithmetic operators:

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

**Operator Result**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator A + B | A + B will give 30 |
| - Subtraction | Subtracts right hand operand from left hand operand A - B | A - B will give -10 |
| \* Multiplication | Multiplies values on either side of the operator A \* B | A \* B will give 200 |
| / Division | Divides left hand operand by right hand operand B / A | B / A will give 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder B % A | B % A will give 0 |

The Relational Operators:

There are following relational operators supported by Java language:

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

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

**The Bitwise Operators**

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. Bitwise operator works on bits and performs bit-by-bit operation. These operators act upon the individual bits of their operands.

Assume if a = 60; and b = 13; now in binary format

they will be as follows:

a = 0011 1100

b = 0000 1101

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

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

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

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it  exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either  operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in  one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and  has the effect of 'flipping' bits. | (~A ) will give -60 which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value  is moved left by the number of bits specified by  the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands  value is moved right by the number of bits  specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> | Shift right zero fill operator. The left operands  value is moved right by the number of bits  specified by the right operand and shifted values  are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

The Logical Operators:

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

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

The Assignment Operators

There are following assignment operators supported by Java language:

|  |  |  |
| --- | --- | --- |
| = | Simple assignment operator, Assigns values  from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right  operand to the left operand and assign the  result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It  subtracts right operand from the left operand  and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies  right operand with the left operand and assign  the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left  operand with the right operand and assign the  result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes  modulus using two operands and assign the  result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

Conditional Operator (?:):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to

evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable.

The operator is written as:

Syntax:

variable x =(expression)? value iftrue: value iffalse

example:

public class Test{

public static void main(String args[]){

int a , b;

a =10;

b =(a ==1)?20:30;

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

b =(a ==10)?20:30;

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

}

}

This would produce the following result:

Value of b is:30

Value of b is:20

**Control Statements**

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java’s program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements, that is, iteration statements form loops. *Jump* statements allow your program to execute in a nonlinear fashion.

1. **Java’s Selection Statements /** Java Decision Making

There are two types of decision making statements in Java. They are:

if statements

switch statements

These statements allow you to control the flow of your program’s execution based upon conditions known only during run time.

1. **if** Statement:

The **if** statement is Java’s conditional branch statement. It can be used to route program execution through two different paths.

Syntax:

if(condition)

{

//Statements will execute if the Boolean expression is true

}

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (if it exists after the closing curly brace) will be executed.

Some programmers find it convenient to include the curly braces when using the **if**, even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don’t have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors.

2. **if...else** Statement:

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

Syntax:

The syntax of an if...else is:

if(Boolean\_expression){

//Executes when the Boolean expression is true

}else{

//Executes when the Boolean expression is false

}

3. **if...else if...else** Statement (Ladder Structure):

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

using single if...else if statement.

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

An if can have zero or one else's and it must come after any else if's.

An if can have zero to many else if's and they must come before the else.

Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else is:

if(Boolean\_expression1){

//Executes when the Boolean expression 1 is true

}elseif(Boolean\_expression2){

//Executes when the Boolean expression 2 is true

}elseif(Boolean\_expression3){

//Executes when the Boolean expression 3 is true

}else{

//Executes when the none of the above condition is true.

}

The **if** statements are executed from the top down. As soon as one of the conditions

controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be

executed. The final **else** acts as a default condition; that is, if all other conditional tests fail,

then the last **else** statement is performed. If there is no final **else** and all other conditions

are **false**, then no action will take place.

4. **Nested if...else** Statement:

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

Syntax:

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

if(Boolean\_expression1){

//Executes when the Boolean expression 1 is true

if(Boolean\_expression2){

//Executes when the Boolean expression 2 is true

}

}

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

**II. The switch Statement**:

The **switch** statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.

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

Syntax:

The syntax of enhanced for loop is:

switch(expression){

case value :

//Statements

break;//optional

case value :

//Statements

break;//optional

//You can have any number of case statements.

default://Optional

//Statements

}

Each value specified in the **case** statements must be a unique constant expression (such as a literal value). Duplicate **case** values are not allowed. The type of each value must be compatible with the type of *expression*.

The **switch** statement works like this: The value of the expression is compared with each of

the values in the **case** statements. If a match is found, the code sequence following that **case**

statement is executed. If none of the constants matches the value of the expression, then the

**default** statement is executed. However, the **default** statement is optional. If no **case** matches

and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a

**break** statement is encountered, execution branches to the first line of code that follows the

entire **switch** statement. This has the effect of “jumping out” of the **switch**.

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

* The variable used in a switch statement can only be a byte, short, int, or char.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
* When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will *fall through*to subsequent cases until a break is reached.
* A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

**Nested switch Statements**

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a

*nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the

**case** constants in the inner **switch** and those in the outer **switch**. For example, the following

fragment is perfectly valid:

switch(count) {

case 1:

switch(target) { // nested switch

case 0:

System.out.println("target is zero");

break;

case 1: // no conflicts with outer switch

System.out.println("target is one");

break;

}

break;

case 2: // ...

Here, the **case 1:** statement in the inner switch does not conflict with the **case 1:** statement

in the outer switch. The **count** variable is compared only with the list of cases at the outer

level. If **count** is 1, then **target** is compared with the inner list cases.

**2. Iteration Statements**

Java’s iteration statements are **for**, **while**, and **do-while**. These statements create what we

commonly call *loops*. As you probably know, a loop repeatedly executes the same set of

instructions until a termination condition is met. As you will see, Java has a loop to fit any

programming need.

**i) For Loop**

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is executed only once. Next, *condition* is evaluated. This must be a Boolean expression.

It usually tests the loop control variable against a target value. If this expression is true, then the

body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the

loop is executed. This is usually an expression that increments or decrements the loop control

variable. The loop then iterates, first evaluating the conditional expression, then executing the

body of the loop, and then executing the iteration expression with each pass. This process

repeats until the controlling expression is false.

**Using the Comma**

There will be times when you will want to include more than one statement in the

initialization and iteration portions of the **for** loop. For example, consider the loop in

the following program:

class Sample {

public static void main(String args[]) {

int a, b;

b = 4;

for(a=1; a<b; a++) {

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

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

b--;

}

}

}

As you can see, the loop is controlled by the interaction of two variables. Since the loop is

governed by two variables, it would be useful if both could be included in the **for** statement,

itself, instead of **b** being handled manually. Fortunately, Java provides a way to accomplish

this. To allow two or more variables to control a **for** loop, Java permits you to include

multiple statements in both the initialization and iteration portions of the **for**. Each

statement is separated from the next by a comma.

Using the comma, the preceding **for** loop can be more efficiently coded, as shown here:

// Using the comma.

class Comma {

public static void main(String args[]) {

int a, b;

for(a=1, b=4; a<b; a++, b--) {

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

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

}

}

}

In this example, the initialization portion sets the values of both **a** and **b**. The two commaseparated

statements in the iteration portion are executed each time the loop repeats. The

program generates the following output:

a = 1

b = 4

a = 2

b = 3

The while Loop

Another of Java’s loops is the **while**. The general form of the **while** loop is

while(*condition*) *statement*;

where *statement* may be a single statement or a block of statements, and *condition* defines the

condition that controls the loop, and it may be any valid Boolean expression. The loop repeats

while the condition is true. When the condition becomes false, program control passes to the

line immediately following the loop.

Here is a simple example in which a **while** is used to print the alphabet:

// Demonstrate the while loop.

class WhileDemo {

public static void main(String args[]) {

char ch;

// print the alphabet using a while loop

ch = 'a';

while(ch <= 'z') {

System.out.print(ch);

ch++;

}

}

}

Here, **ch** is initialized to the letter a. Each time through the loop, **ch** is output and then

incremented. This process continues until **ch** is greater than z.

As with the **for** loop, the **while** checks the conditional expression at the top of the loop,

which means that the loop code may not execute at all. This eliminates the need for performing

a separate test before the loop. The following program illustrates this characteristic of the

**while** loop. It computes the integer powers of 2, from 0 to 9.

// Compute integer powers of 2.

class Power {

public static void main(String args[]) {

int e;

int result;

for(int i=0; i < 10; i++) {

result = 1;

e = i;

while(e > 0) {

result \*= 2;

e--;

}

System.out.println("2 to the " + i +

" power is " + result);

}

}

}

The output from the program is shown here:

2 to the 0 power is 1

2 to the 1 power is 2

2 to the 2 power is 4

2 to the 3 power is 8

2 to the 4 power is 16

2 to the 5 power is 32

2 to the 6 power is 64

2 to the 7 power is 128

2 to the 8 power is 256

2 to the 9 power is 512

***do...while* Loops**

The do loop is just like a while loop, except that do executes a given statement or block until a

condition is false. The main difference is that while loops test the condition before looping,

making it possible that the body of the loop will never execute if the condition is false the first

time it’s tested. do loops run the body of the loop at least once before testing the condition. do

loops look like this:

do {

bodyOfLoop;

} while (condition);

Here, the bodyOfLoop part is the statements that are executed with each iteration. It’s shown here

with a block statement because it’s most commonly used that way, but you can substitute the

braces for a single statement as you can with the other control-flow constructs. The condition

is a boolean test. If it returns true, the loop is run again. If it returns false, the loop exits. Keep

in mind that with do loops, the body of the loop executes at least once.

Here’s a simple example of a do loop that prints a message each time the loop iterates:

int x = 1;

do {

System.out.println(“Looping, round “ + x);

x++;

} while (x <= 10);

Here’s the output of these statements:

Looping, round 1

Looping, round 2

Looping, round 3

Looping, round 4

Looping, round 5

Looping, round 6

Looping, round 7

Looping, round 8

Looping, round 9

Looping, round 10

**3.** **Jump Statements**

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

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

## Syntax

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

break;

## Example

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

if( x == 30 ) {

break;

}

System.out.print( x );

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

10

20

**Continue Statement**

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

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

## Syntax

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

continue;

## Example

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

if( x == 30 ) {

continue;

}

System.out.print( x );

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

10

20

40

50

AJavamethod is a collection of statements that are grouped together to perform an operation. When you

call the System.out.println method, for example, the system actually executes several statements in order to display

a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without

parameters, overload methods using the same names, and apply method abstraction in the program design.

Creating

a

Method:

In general, a method has the following syntax:

modifier returnValueType methodName(list of parameters){

// Method body;

}

A method definition consists of a method header and a method body. Here are all the parts of a method:

**Modifiers:** The modifier, which is optional, tells the compiler how to call the method. This defines the access

type of the method.

**Return Type:** A method may return a value. The returnValueType is the data type of the value the method

returns. Some methods perform the desired operations without returning a value. In this case, the

returnValueType is the keyword **void**.

**Method Name:** This is the actual name of the method. The method name and the parameter list together

constitute the method signature.

**Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the

parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type,

order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no

parameters.

**Method Body:** The method body contains a collection of statements that define what the method does.

Example:

Here is the source code of the above defined method called max(). This method takes two parameters num1 and

num2 and returns the maximum between the two:

/\*\* Return the max between two numbers \*/

public static int max(int num1,int num2){

int result;

if(num1 > num2)

result = num1;

else

result = num2;

return result;

}

Calling

a

Method:

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke

it. There are two ways to call a method; the choice is based on whether the method returns a value or not.

When a program calls a method, program control is transferred to the called method. A called method returns

control to the caller when its return statement is executed or when its method-ending closing brace is reached.

If the method returns a value, a call to the method is usually treated as a value. For example:

int larger = max(30,40);

If the method returns void, a call to the method must be a statement. For example, the method println returns void.

The following call is a statement:

System.out.println("Welcome to Java!");

Example:

Following is the example to demonstrate how to define a method and how to call it:

public class TestMax{

/\*\* Main method \*/

public static void main(String[] args){

int i =5;

int j =2;

int k = max(i, j);

System.out.println("The maximum between "+ i +

" and "+ j +" is "+ k);

}

/\*\* Return the max between two numbers \*/

public static int max(int num1,int num2){

int result;

if(num1 > num2)

result = num1;

else

result = num2;

return result;

}

}

This would produce the following result:

The maximum between 5and2is5

This program contains the main method and the max method. The main method is just like any other method except

that it is invoked by the JVM.

The main method's header is always the same, like the one in this example, with the modifiers public and static,

return value type void, method name main, and a parameter of the String[] type. String[] indicates that the parameter

is an array of String**.**

Passing

Parameters

by

Values:

When calling a method, you need to provide arguments, which must be given in the same order as their respective

parameters in the method specification. This is known as parameter order association.

For example, the following method prints a message n times:

public static void nPrintln(String message,int n){

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

System.out.println(message);

}

Here, you can use nPrintln("Hello", 3) to print "Hello" three times. The nPrintln("Hello", 3) statement passes the

actual string parameter, "Hello", to the parameter, message; passes 3 to n; and prints "Hello" three times. However,

the statement nPrintln(3, "Hello") would be wrong.

When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred

to as pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the

parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means passing

the value of x to y.

Example:

Following is a program that demonstrates the effect of passing by value. The program creates a method for

swapping two variables. The swap method is invoked by passing two arguments. Interestingly, the values of the

arguments are not changed after the method is invoked.

public class TestPassByValue{

public static void main(String[] args){

int num1 =1;

int num2 =2;

System.out.println("Before swap method, num1 is "+num1 +" and num2 is "+ num2);

// Invoke the swap method

swap(num1, num2);

System.out.println("After swap method, num1 is "+num1 +" and num2 is "+ num2);

}

/\*\* Method to swap two variables \*/

public static void swap(int n1,int n2){

System.out.println("\tInside the swap method");

System.out.println("\t\tBefore swapping n1 is "+ n1+" n2 is "+ n2);

// Swap n1 with n2

int temp = n1;

n1 = n2;

n2 = temp;

System.out.println("\t\tAfter swapping n1 is "+ n1+" n2 is "+ n2);

}

}

This would produce the following result:

Before swap method, num1 is1and num2 is2

Inside the swap method

Before swapping n1 is1 n2 is2

After swapping n1 is2 n2 is1

After swap method, num1 is1and num2 is2

Overloading

Methods:

The max method that was used earlier works only with the int data type. But what if you need to find which of two

floating-point numbers has the maximum value? The solution is to create another method with the same name but

different parameters, as shown in the following code:

public static double max(double num1,double num2){

if(num1 > num2)

return num1;

else

return num2;

}

If you call max with int parameters, the max method that expects int parameters will be invoked; if you call max with

double parameters, the max method that expects double parameters will be invoked. This is referred to as **method**

**overloading**; that is, two methods have the same name but different parameter lists within one class.

The Java compiler determines which method is used based on the method signature. Overloading methods can

make programs clearer and more readable. Methods that perform closely related tasks should be given the same

name.

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers

or return types. Sometimes there are two or more possible matches for an invocation of a method due to similar

method signature, so the compiler cannot determine the most specific match. This is referred to as ambiguous

invocation**.**

The

finalize(

)

Method:

It is possible to define a method that will be called just before an object's final destruction by the garbage collector.

This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever

it is about to recycle an object of that class.

Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

The finalize( ) method has this general form:

protected void finalize()

{

// finalization code here

}

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.

This means that you cannot know whenor even iffinalize( ) will be executed. For example, if your program ends

before garbage collection occurs, finalize( ) will not execute.

**Constructors**

**Constructor in java** is a *special type of method* that is used to initialize the object. This can also be called creating an instance. 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.

### The constructor return any value that is current class instance. One cannot use return type yet it returns a value. The constructor can perform other tasks instead of initialization, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

Rules for creating java constructor

### Constructor name must be same as its class name

### Constructor is invoked implicitly.

### Constructor must have no explicit return type

### The java compiler provides a default constructor if you don't have any constructor

**Types of java constructors**

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor
3. **Default constructor:**

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

If you do not define any constructor in your class, java generates one for you by default. This constructor is known as default constructor. You would not find it in your source code but it would present there.

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

**Syntax:**

<class\_name>()

{

}

Example:

class Constru{

Bike1()

{

System.out.println("Default constructor is created");

}

public static void main(String args[])

{

Bike1 b=new Bike1();

}

}

Output:

Default constructor is created

**Parameterized constructor**

A constructor that have parameters is known as parameterized constructor.

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

class Example2

{

private int var;

public Example2(int num)

{

//code for parameterized one

var = num;

}

public int getValue()

{

return var;

}

public static void main(String args[])

{

Example2 obj2 = new Example2(10);

System.out.println("var is: "+obj2.getValue());

}

}

Output:

var is: 10

## 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

class Student5{

    int id;

    String name;

    int age;

    Student5(int i,String n){

    id = i;

    name = n;

    }

    Student5(int i,String n,int a){

    id = i;

    name = n;

    age=a;

    }

    void display(){System.out.println(id+" "+name+" "+age);

}

      public static void main(String args[]){

    Student5 s1 = new Student5(111,"Karan");

    Student5 s2 = new Student5(222,"Aryan",25);

    s1.display();

    s2.display();

   }

}

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. |

## 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 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.

class Student8{

   int rollno;

   String name;

   static String college ="ITS";

   Student8(int r,String n){

   rollno = r;

   name = n;

   }

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

 public static void main(String args[]){

 Student8 s1 = new Student8(111,"Karan");

 Student8 s2 = new Student8(222,"Aryan");

 s1.display();

 s2.display();

 }

}

Output:

111 Karan ITS

222 Aryan ITS

### // 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. |

class Counter2{

static int count=0;//will get memory only once and retain its value

Counter2(){

count++;

System.out.println(count);

}

  public static void main(String args[]){

Counter2 c1=new Counter2();

Counter2 c2=new Counter2();

Counter2 c3=new Counter2();

   }

}

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.

**Restrictions for static method**

|  |
| --- |
| 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.  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 //Program of changing the common property of all objects(static field).    class Student9{       int rollno;       String name;       static String college = "ITS";         static void change(){       college = "BBDIT";       }       Student9(int r, String n){       rollno = r;       name = n;       }      void display (){System.out.println(rollno+" "+name+" "+college);}       public static void main(String args[]){      Student9.change();       Student9 s1 = new Student9 (111,"Karan");      Student9 s2 = new Student9 (222,"Aryan");      Student9 s3 = new Student9 (333,"Sonoo");       s1.display();      s2.display();      s3.display();      }  }  Output:111 Karan BBDIT  222 Aryan BBDIT  333 Sonoo BBDIT 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

class A2{

  static{System.out.println("static block is invoked");}

  public static void main(String args[]){

   System.out.println("Hello main");

  }

}

Output:static block is invoked

Hello main

**Arrays**

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.



### Advantage of Java Array

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
* **Random access:** We can get any data located at any index position.

**One-Dimensional Arrays**

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

**Syntax:**

*type var-name*[ ];

For example,

String students\_name[];

The above declares an array named **students\_name** with the type “array of String”.

### Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

class Testarray{

public static void main(String args[]){

int a[]=new int[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//printing array

for(int i=0;i<a.length;i++)//length is the property of array

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

}}

Output: 10

20

70

40

50

**Multidimensional Arrays**

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look

and act like regular multidimensional arrays. However, as you will see, there are a couple

of subtle differences. To declare a multidimensional array variable, specify each additional

index using another set of square brackets. For example, the following declares a twodimensional

array variable called **twoD**:

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

This allocates a 4 by 5 array and assigns it to **twoD**. Internally, this matrix is implemented as

an *array* of *arrays* of **int**.

The following program numbers each element in the array from left to right, top to

bottom, and then displays these values:

// Demonstrate a two-dimensional array.

class TwoDArray {

public static void main(String args[]) {

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

int i, j, k = 0;

for(i=0; i<4; i++)

for(j=0; j<5; j++) {

twoD[i][j] = k;

k++;

}

for(i=0; i<4; i++) {

for(j=0; j<5; j++)

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

System.out.println();

}

}

}

This program generates the following output:

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

### Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

class Testarray5{

public static void main(String args[]){

//creating two matrices

int a[][]={{1,3,4},{3,4,5}};

int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices

int c[][]=new int[2][3];

//adding and printing addition of 2 matrices

for(int i=0;i<2;i++){

for(int j=0;j<3;j++){

c[i][j]=a[i][j]+b[i][j];

System.out.print(c[i][j]+" ");

}

System.out.println();//new line

}

}}

Output:2 6 8

6 8 10

The

foreach

Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the

complete array sequentially without using an index variable.

Example:

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

public class TestArray{

public static void main(String[] args){

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

// Print all the array elements

for(double element: myList){

System.out.println(element);

}

}

}

This would produce the following result:

1.9

2.9

3.4

3.5

# Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

## [What Is an Object?](https://docs.oracle.com/javase/tutorial/java/concepts/object.html)

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

## [What Is a Class?](https://docs.oracle.com/javase/tutorial/java/concepts/class.html)

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

## [What Is Inheritance?](https://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html)

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

## [What Is an Interface?](https://docs.oracle.com/javase/tutorial/java/concepts/interface.html)

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

## [What Is a Package?](https://docs.oracle.com/javase/tutorial/java/concepts/package.html)

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

**Class:**

The class forms the basis for object-oriented programming in Java.

Any concept you wish to implement in a Java program must be encapsulated within a class.

A class is that it defines a new data type. Once defined, this new type can be used to create objects of that type.

Thus, a class is a *template* for an object, and an object is an *instance* of a class.

**Syntax:**

class *classname* {

*type instance-variable1;*

**1. Inheritance**

**Inheritance** is one of the feature of Object-Oriented Programming ([OOPs](http://beginnersbook.com/2013/04/oops-concepts/)).

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

It allows the creation of hierarchical classifications.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

Inheritance is a [compile-time](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) mechanism. A super-class can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritance.

Advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.

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

**Syntax**

class Super {

.....

.....

}

class Sub extends Super {

.....

.....

}

**Access Modifiers in JAVA**

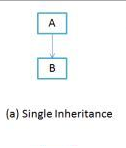
The derived class inherits all the members and methods that are declared as public or protected. If declared as private it can not be inherited by the derived classes. The private members can be accessed only in its own class. The private members can be accessed through assessor methods as shown in the example below. The derived class cannot inherit a member of the base class if the derived class declares another member with the same name.

## Types of inheritance in java

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

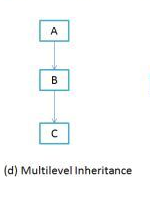
**1. Single Inheritance**

**Single inheritance** is Very easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



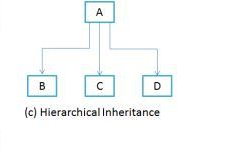
### 2. Multilevel Inheritance

**Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.



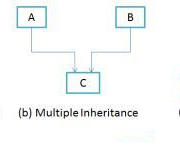
### 3. Hierarchical Inheritance

In such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.



### 4. Multiple Inheritance

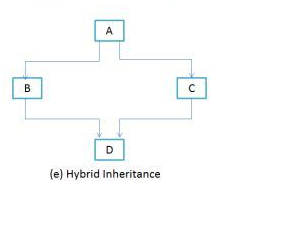
“**Multiple Inheritance**” refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes. Using multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

[](http://beginnersbook.com/wp-content/uploads/2013/05/Multiple-Inheritance.png)

Note : Most of the new OO languages like **Small Talk, Java, C# do not support Multiple inheritance**. Multiple Inheritance is supported in C++.

### 5. Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single, Hierarchical** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance ! Using interfaces. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.

[](http://beginnersbook.com/wp-content/uploads/2013/05/Hybrid-inheritance.png)

# this keyword

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.

**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. |

//example of this keyword

class Student11{

    int id;

    String name;

    Student11(int id,String name){

    this.id = id;

    this.name = name;

    }

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

    public static void main(String args[]){

    Student11 s1 = new Student11(111,"Karan");

    Student11 s2 = new Student11(222,"Aryan");

    s1.display();

    s2.display();

}

}

//example of this keyword

class Student11{

    int id;

    String name;

    Student11(int id,String name){

    this.id = id;

    this.name = name;

    }

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

    public static void main(String args[]){

    Student11 s1 = new Student11(111,"Karan");

    Student11 s2 = new Student11(222,"Aryan");

    s1.display();

    s2.display();

}

}

Output111 Karan

222 Aryan

### 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.

//Program of this() constructor call (constructor chaining)

class Student13{

    int id;

    String name;

    Student13(){System.out.println("default constructor is invoked");}

    Student13(int id,String name){

    this ();//it is used to invoked current class constructor.

    this.id = id;

    this.name = name;

    }

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

    public static void main(String args[]){

    Student13 e1 = new Student13(111,"karan");

    Student13 e2 = new Student13(222,"Aryan");

    e1.display();

    e2.display();

   }

}

Output:

default constructor is invoked

default constructor is invoked

111 Karan

222 Aryan

### 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.

class S{

  void m(){

  System.out.println("method is invoked");

  }

  void n(){

  this.m();//no need because compiler does it for you.

  }

  void p(){

  n();//complier will add this to invoke n() method as this.n()

  }

  public static void main(String args[]){

  S s1 = new S();

  s1.p();

  }

}

class S{

  void m(){

  System.out.println("method is invoked");

  }

  void n(){

  this.m();//no need because compiler does it for you.

  }

  void p(){

  n();//complier will add this to invoke n() method as this.n()

  }

  public static void main(String args[]){

  S s1 = new S();

  s1.p();

  }

}

Output:method is invoked

# 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.

super is used to refer immediate parent class instance variable.

//example of super keyword

class Vehicle{

  int speed=50;

}

class Bike4 extends Vehicle{

  int speed=100;

  void display(){

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

  }

  public static void main(String args[]){

   Bike4 b=new Bike4();

   b.display();

}

}

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: |

class Vehicle{

  Vehicle(){System.out.println("Vehicle is created");}

}

class Bike5 extends Vehicle{

  Bike5(){

   super();//will invoke parent class constructor

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

  }

  public static void main(String args[]){

   Bike5 b=new Bike5();

}

}

Output:Vehicle is created

Bike is created

## 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: |

class Person{

void message(){System.out.println("welcome");}

}

class Student16 extends Person{

void message(){System.out.println("welcome to java");}

void display(){

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

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

}

public static void main(String args[]){

Student16 s=new Student16();

s.display();

}

}

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. |

**Interfaces**

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.

[abstract class](http://beginnersbook.com/2013/05/java-abstract-class-method/) is used to achieve partial abstraction(hiding irrelevant details from user). But the interfaces are used for achieving full abstraction.

#### The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

An interface is similar to a class in the following ways −

* An interface can contain any number of methods.
* An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
* The byte code of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

In this post we will discuss **difference between Abstract Class and Interface in Java with examples.**

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

**Declaration**  
Interfaces are declared by specifying a keyword “interface”. E.g.:

interface MyInterface

{

/\* All the methods are public abstract by default

\* Note down that these methods are not having body

\*/

public void method1();

public void method2();

}