**Object oriented Programming**

**Unit - 1**

**History of Java :**

**The history of Java** is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of java starts with Green Team. Java team members (also known as **Green Team**), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted and Dynamic".

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. There are given the significant points that describe the history of Java.

1) [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java), **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java named "Oak"?

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

## Why Java Programming named "Java"?

7) **Why had they chosen java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at [Sun Microsystems](https://www.javatpoint.com/sun-microsystems) (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 released in(January 23, 1996).

**Evolution of JAVA:**

The development of each programming language is based on a fact: there is a need to solve a problem that was not resolved by previous programming languages. Early programmers had to choose different programming languages, usually for various tasks, such as a specific language for a type of field. A certain language was sufficient to solve the problems of its field but was not able to solve the problems of other fields. For example, **Fortran**could have been used to write efficient programs for scientific problems, but it was not good for system code. Similarly, **Basic** was easy to understand but was not robust to write big programs; While the **assembly language** was powerful for writing efficient programs, but it was not easy to remember and execution.

Programming languages such as **Cobol, Fortran** do not have structural principles. They use the Goto statement to control the flow of the program. Therefore, programs using this type of code are made up of many jumps and conditional statements that make it difficult to understand.

Therefore,[**C**](https://www.w3schools.in/c-tutorial/) was invented in 1970, to replace the assembly language and to create a structured, effective and high-level language. The development of C was the result of the development process started with **BCPL** by Dennis Ritchie. BCPL is an old language developed by Martin Richard.  Ken Thompson developed a language called **B**, which was influenced by BCPL.

**C** is a processor-oriented programming language; it is easy to execute and understand. C became quite famous at that time because it was reliable, simple and easy to use.

Though **C** was quite efficient and successful programming language, the complexity of the program was seeking more efficient language to solve problems. When we write a program in C, it has a limit, such as a maximum of 25000 lines of code, beyond which it cannot handle the complexity. But writing and managing large programs was a demand at that time. So a new concept came.

[**C++**](https://www.w3schools.in/cplusplus-tutorial/) came with object-oriented programming features. C++ is the extension of C language which has been used extensively. It is a powerful modern language that includes the power and simplicity of C and the characteristics of OOP. C++ provides more functional software benefits than C.

C ++ with OOP became quite famous but then a new problem arose, to control the software on different machines, a separate compiler is required for that CPU. But building a C++ compiler was quite expensive. Therefore, an efficient and easy solution was needed, and this requirement became the reason for the creation of **Java, which is a portable and platform-independent language.**

**Java version Release History:**

|  |  |
| --- | --- |
| JDK1.0 | 23 Jan 1996 |
| JDK1.1 | 19 Feb 1997 |
| J2SE 1.2 | 8 Dec 1998 |
| J2SE 1.3 | 8 May 2000 |
| J2SE 1.4 | 6 Feb 2002 |
| J2SE 5.0 | 30 Sept 2004 |
| Java SE 6 | 11 Dec 2006 |
| Java SE 7.0 | 28 July 2011 |
| Java SE 8.0 | 18 March 2014 |
| Java SE 9.0 | Sept 2017 |
| Java SE 10 (18.3) | 2018 |

**Java’s Magic: The Byte code**

The key that allows java to solve both the security and the portability problems just described is that the output of a java compiler is not executable code. Rather, it is byte code, byte code is a highly optimized set of instructions designed to be executed by the java runtime system, which is called the java virtual machine (JVM). That is, in its standard form, the JVM is an interpreter for byte code, this may come as a bit of a surprise. As you know, C+++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted mostly because of performance concerns. However, the fact that a java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why.

Translating a java program into bytecode helps makes it much easier ot run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform, all interpret the same java bytecode. If a java program were compiled to native code, the different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs.

The fact that a java program is interpreted also helps to make it secure. Because the execution of every java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the java language.

When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with java, the differential between the two is not so great. The use of bytecode enables the java run-time system to execute programs much faster than you might expect.

Although java was designed for interpretation, there is technically nothing about java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire java program into executable code all at once, because java performs various run-time checks that can be done only at run time. Instead, the JIT compiles code, as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the run-time system (which performs the compilation) still is in charge of the execution environment. Whether your java program is actually interpreted in the traditional way or compiled on-the-fly, its functionality is the same.

**Servlets: Java on the servers side:**

[**Servlets**](https://www.roseindia.net/servlets/index.shtml) are preferred by developers who are working on developing Server side programs and applications as it has platform-independent methods and do not have the limitations of CGI programs. This makes the programs and application fast and efficient.

[**Servlets**](https://www.roseindia.net/servlets/index.shtml) are mostly used to develop interactive Web-based applications that can run on any servlet enabled web server. **Servlets use of Java language**and a standard framework makes it the first choice for developers to create sophisticated server side extensions.

You can use Java servlets to

1. generate dynamic pages,
2. read from and write to databases,
3. maintain a consistent user state as your user moves from page to page.

servlet also interact with

1. HTML forms,
2. client-side Java applets, and
3. databases.

**Features of Java (or) Java Buzz words :**

The primary objective of [Java programming](https://www.javatpoint.com/java-tutorial) language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.



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

### Simple :

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

### Object-oriented :

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. 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 simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. [Object](https://www.javatpoint.com/object-and-class-in-java)
2. Class
3. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
4. [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
5. [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
6. [Encapsulation](https://www.javatpoint.com/encapsulation)

### Platform Independent



Java is platform independent because it is different from other languages like [C](https://www.javatpoint.com/c-programming-language-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. 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 a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms, for example, 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., Write Once and Run Anywhere (WORA).

### Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

* **No explicit pointer**
* **Java Programs run inside a virtual machine sandbox**



* **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
* **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
* **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

### Robust

Robust simply means strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

### Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

### Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

### High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

### Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to 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 doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

### Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

**C++ vs Java**

There are many differences and similarities between the [C++ programming](https://www.javatpoint.com/cpp-tutorial) language and [Java](https://www.javatpoint.com/java-tutorial). A list of top differences between C++ and Java are given below:

|  |  |  |
| --- | --- | --- |
| **Comparison Index** |  **C++** |  **Java** |
| **Platform-independent** | C++ is platform-dependent. | Java is platform-independent. |
| **Mainly used for** | C++ is mainly used for system programming. | Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications. |
| **Design Goal** | C++ was designed for systems and applications programming. It was an extension of [C programming language](https://www.javatpoint.com/c-programming-language-tutorial). | Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience. |
| **Goto** | C++ supports the [goto](https://www.javatpoint.com/cpp-goto-statement) statement. | Java doesn't support the goto statement. |
| **Multiple inheritance** | C++ supports multiple inheritance. | Java doesn't support multiple inheritance through class. It can be achieved by [interfaces in java](https://www.javatpoint.com/interface-in-java). |
| **Operator Overloading** | C++ supports [operator overloading](https://www.javatpoint.com/cpp-overloading). | Java doesn't support operator overloading. |
| **Pointers** | C++ supports [pointers](https://www.javatpoint.com/cpp-pointers). You can write pointer program in C++. | Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java. |
| **Compiler and Interpreter** | C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent. | Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent. |
| **Call by Value and Call by reference** | C++ supports both call by value and call by reference. | Java supports call by value only. There is no call by reference in java. |
| **Structure and Union** | C++ supports structures and unions. | Java doesn't support structures and unions. |
| **Thread Support** | C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support. | Java has built-in [thread](https://www.javatpoint.com/multithreading-in-java) support. |
| **Documentation comment** | C++ doesn't support documentation comment. | Java supports documentation comment (/\*\* ... \*/) to create documentation for java source code. |
| **Virtual Keyword** | C++ supports virtual keyword so that we can decide whether or not override a function. | Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default. |
| **unsigned right shift >>>** | C++ doesn't support >>> operator. | Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator. |
| **Inheritance Tree** | C++ creates a new inheritance tree always. | Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the [inheritance](https://www.javatpoint.com/inheritance-in-java) tree in java. |
| **Hardware** | C++ is nearer to hardware. | Java is not so interactive with hardware. |
| **Object-oriented** | C++ is an object-oriented language. However, in C language, single root hierarchy is not possible. | Java is also an [object-oriented](https://www.javatpoint.com/java-oops-concepts) language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object. |

**Java Syntax:**

MyClass.java

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

### Example explained :

Every line of code that runs in Java must be inside a class. In our example, we named the class **MyClass**. A class should always start with an uppercase first letter.

**Note:** Java is case-sensitive: "Hello" and "hello" has different meaning.

The name of the java file **must match** the class name. When saving the file, save it using the class name and add ".java" to the end of the filename. To run the example above on your computer, make sure that Java is properly installed. The output should be:

Output: Hello World

## The main Method

The main() method is required and you will see it in every Java program:

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

Any code inside the main() method will be executed. You don't have to understand the keywords before and after main. You will get to know them bit by bit while reading this tutorial.

For now, just remember that every Java program has a class name which must match the filename, and that every program must contain the main() method.

## System.out.println()

Inside the main() method, we can use the println() method to print a line of text to the screen:

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

**Note:** In Java, each code statement must end with a semicolon.

## Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line comments starts with two forward slashes (//).

Any text between // and the end of the line is ignored by Java (will not be executed).

This example uses a single-line comment before a line of code:

**Example**

// This is a comment
System.out.println("Hello World");

## Java Multi-line Comments

Multi-line comments start with /\* and ends with \*/.

Any text between /\* and \*/ will be ignored by Java.

This example uses a multi-line comment (a comment block) to explain the code:

**Example**

/\* The code below will print the words Hello World
to the screen, and it is amazing \*/
System.out.println("Hello World");

**Lexical Issues**

Now we will see atomic elements of java. Java programs are a collection of white space identifiers, comments, literals, operators, separators, and keywords.

**Whitespace**

            Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In java, whitespace is a *space, tab, or new line.*

**Identifiers**

             Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of *uppercase and lowercase letters, numbers or the underscore and dollar sign characters*. They must not begin with a number, in case they be confused with a numeric literal. Again, java is case-sensitive, so **VALUE**is a different identifier the**Value.**Some examples of valid identifiers are:

 AvgTemp            count            a4            $test            this\_is\_ok

 Invalid variable names include:

 2count   high-temp      Not/ok

**Literals**

 Using a literal representation of it creates a constant value in java. For example, here are some literals:

 100       98.6            ‘X’            “This is a test”

             Left to right, the first literal specifies an *integer*, the next is a *floating-point value*, the third is a *character constant*, and the last is a *string*. A literal can be used anywhere a value of its type is allowed.

**Comments**

             As mentioned, there are three types of comments defined by java. You have already seen two: *single-line and multiline*. The third type is called a *documentation comment*. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /\*\* and ends with a\*/.

**Advantages of Library Classes in Java:**

* The type constraints are used to control where the java library classes can be replaced with routine versions without affecting type perfection of programs.
* Static analysis is then used to control those applicants for which unused library functionality and synchronization can be removed safely from the allocated types.
* The profile data is collected about the usage features of the customization candidates to determine where the allocation of custom library classes is likely to be cost-effective.
* To base on the static analysis results and the profiling information the custom library classes are automatically generated from a template.
* The bytecode of the client application is rewritten to use the generated custom classes. This bytecode rewriting is completely see-through to the programmer.

### List of Library Classes in Java:

|  |  |
| --- | --- |
| Library classes | Purpose of the class |
| Java.io | Use for input and output functions. |
| Java.lang | Use for character and string operation. |
| Java.awt | Use for windows interface. |
| Java.util | Use for develop utility programming. |
| Java.applet | Use for applet. |
| Java.net | Used for network communication. |
| Java.math | Used for various mathematical calculations like power, square root etc. |

**Data types, Arrays and variables:**

**Java Variables**

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

* String - stores text, such as "Hello". String values are surrounded by double quotes
* int - stores integers (whole numbers), without decimals, such as 123 or -123
* float - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* boolean - stores values with two states: true or false

### Example :

Create a variable called **name** of type String and assign it the value "**Sri Lakshmi**":

String name = " **Sri Lakshmi**";

System.out.println(name);

Create a variable called **myNum** of type int and assign it the value **15**:

int myNum = 15;
System.out.println(myNum);

You can also declare a variable without assigning the value, and assign the value later:

**Example**

int myNum;
myNum = 15;
System.out.println(myNum);

A demonstration of how to declare variables of other types:

### Example :

int myNum = 5;
float myFloatNum = 5.99f;
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello";

**Display Variables**

The println() method is often used to display variables.

To combine both text and a variable, use the + character:

String name = "John";
System.out.println("Hello " + name);

You can also use the + character to add a variable to another variable:

### Example

String firstName = "John ";
String lastName = "Doe";
String fullName = firstName + lastName;
System.out.println(fullName);

For numeric values, the + character works as a mathematical operator (notice that we use int (integer) variables here):

**Example**

int x = 5;
int y = 6;
System.out.println(x + y); // Print the value of x + y

## Declare Many Variables:

To declare more than one variable of the **same type**, use a comma-separated list:

**Example**

int x = 5, y = 6, z = 50;
System.out.println(x + y + z);

**Java Identifiers :**

All Java **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

* Names can contain letters, digits, underscores, and dollar signs
* Names should begin with a letter
* Names can also begin with $ and \_ (but we will not use it in this tutorial)
* Names are case sensitive ("myVar" and "myvar" are different variables)
* Names should start with a lowercase letter and it cannot contain whitespace
* Reserved words (like Java keywords, such as int or String) cannot be used as names

**Java Data Types :**

Data types are divided into two groups:

* Primitive data types - includes byte, short, int, long, float, double, boolean and char
* Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp)

**Primitive Data Types**

* A primitive data type specifies the size and type of variable values, and it has no additional methods.
* There are eight primitive data types in Java:

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223.372,036.854,775.808 to 9,223.372,036,854,775,808 |
| float | 4 bytes | Stores fractional numbers from 3.4e−038 to 3.4e+038. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers from 1.7e−308 to 1.7e+038. Sufficient for storing 15 decimal digits |
| boolean | 1 byte | Stores true or false values |
| char | 2 bytes | Stores a single character/letter |
|  |  |  |

**Numbers :**

Primitive number types are divided into two groups:

**Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.

**Floating point types** represents numbers with a fractional part, containing one or more decimals. There are two types: float and double.

Even though there are many numeric types in Java, the most used for numbers are int (for whole numbers) and double (for floating point numbers).

## Integer Types

### Byte

The byte data type can store whole numbers from -128 and 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

**Example**

byte myNum =  100;
System.out.println(myNum);

### Short

The short data type can store whole numbers from -32768 to 32767:

### Example

short myNum = 5000;
System.out.println(myNum);

### Int

The int data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

### Example

int myNum =  100000;
System.out.println(myNum);

### Long

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775808. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

### Example

long myNum = 15000000000L;
System.out.println(myNum);

## Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

### Float

The float data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "f":

### Example

float myNum= 5.75f;
System.out.println(myNum);

### Double

The double data type can store fractional numbers from 1.7e−308 to 1.7e+038. Note that you should end the value with a "d":

### Example

double myNum= 19.99d;
System.out.println(myNum);

Use float or double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of float is only six or seven decimal digits, while double variables have a precision of about 15 digits. Therefore it is safer to use double for most calculations.

### Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

**Example**

float f1= 35e3f;
double d1= 12E4d;
System.out.println(f1);
System.out.println(d1);

## Booleans

A boolean data type is declared with the boolean keyword and can only take the values true or false:

### Example

boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);     // Outputs true
System.out.println(isFishTasty);   // Outputs false

Boolean values are mostly used for conditional testing.

## Characters

The char data type is used to store a **single** character. A char value must be surrounded by single quotes, like 'A' or 'c':

### Example

char myGrade = 'B';
System.out.println(myGrade);

## Strings

The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

### Example

String greeting = "Hello World";
System.out.println(greeting);

The String data type is so much used and integrated in Java, that some call it "the special **ninth** type".

A String in Java is actually a **non-primitive** data type, because it refers to an object. The String object has methods that is used to perform certain operations on strings. **Don't worry if you don't understand the term "object" just yet**. We will learn more about strings and objects in a later chapter.

**Non-Primitive Data Types**

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between primitive and non-primitive data types are:

* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitve types can be **null**.
* A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are String, Arrays, Classes, Interface, etc.

**Type conversion and casting:**

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

**Widening or Automatic Type Conversion :**

Widening conversion takes place when two data types are automatically converted. This happens when:

* The two data types are compatible.
* When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.


Example:

|  |
| --- |
| class Test {     public static void main(String[] args)     {         int i = 100;                    //automatic type conversion         long l = i;                    //automatic type conversion         float f = l;          System.out.println("Int value "+i);         System.out.println("Long value "+l);         System.out.println("Float value "+f);     } }  |

**Output:**

Int value 100

Long value 100

Float value 100.0

**Narrowing or Explicit Conversion**

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

* This is useful for incompatible data types where automatic conversion cannot be done.
* Here, target-type specifies the desired type to convert the specified value to.



char and number are not compatible with each other. Let’s see when we try to convert one into other.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| //Java program to illustrate incompatible data  // type for explicit type conversion public class Test {   public static void main(String[] argv)   {     char ch = 'c';     int num = 88;     ch = num;   } }  |

**Error:**

7: error: incompatible types: possible lossy conversion from int to char

 ch = num;

 ^

1 error

**How to do Explicit Conversion?**
Example:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| //Java program to illustrate explicit type conversion class Test {     public static void main(String[] args)     {         double d = 100.04;          //explicit type casting         long l = (long)d;         //explicit type casting          int i = (int)l;          System.out.println("Double value "+d);         //fractional part lost         System.out.println("Long value "+l);          //fractional part lost         System.out.println("Int value "+i);      }  }  |

Output:

Double value 100.04

Long value 100

Int value 100

**Automatic Type promotion in Expressions**

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Example:

//Java program to illustrate Type promotion in Expressions

class Test

{

    public static void main(String args[])

    {

        byte b = 42;

        char c = 'a';

        short s = 1024;

        int i = 50000;

        float f = 5.67f;

        double d = .1234;

        // The Expression

        double result = (f \* b) + (i / c) - (d \* s);

        //Result after all the promotions are done

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

    }

}

Output:

Result = 626.7784146484375

**Explicit type casting in Expressions**

While evaluating expressions, the result is automatically updated to larger data type  of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.
Example:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| //Java program to illustrate type casting int to byte class Test {      public static void main(String args[])      {          byte b = 50;                    //type casting int to byte         b = (byte)(b \* 2);          System.out.println(b);     } }  |

Output : **100**

# Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

**Access the Elements of an Array**

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

### Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo

**Note:** Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

**Change an Array Element**

To change the value of a specific element, refer to the index number:

### Example

cars[0] = "Opel";

### Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo

## Array Length

To find out how many elements an array have, use the length property:

## Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4

## Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

**Example**

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
**Loop Through an Array with For-Each**

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays

**Syntax**

for (type variable : arrayname) {
  ...
}
The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

**Example**

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
The example above can be read like this: **for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of**i**.

If you compare the for loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

## Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

**Example**

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

**myNumbers** is now an array with two arrays as its elements.

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

**Example**

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x); // Outputs 7

We can also use a for loop inside another for loop to get the elements of a two-dimensional array (we still have to point to the two indexes):

**Example**

public class MyClass {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}

**Widening Primitive Conversion in Java :**

Here is a small code snippet given. Try to Guess the output

|  |
| --- |
| public class Test {     public static void main(String[] args)     {         System.out.print("Y" + "O");         System.out.print('L' + 'O');     } }  |

*At first glance, we expect “YOLO” to be printed.*

**Actual Output:**
“YO155”.

**Explanation:**
When we use double quotes, the text is treated as a string and “YO” is printed, but when we use single quotes, the characters ‘L’ and ‘O’ are converted to int. This is called widening primitive conversion. After conversion to integer, the numbers are added ( ‘L’ is 76 and ‘O’ is 79) and 155 is printed.

Now try to guess the output of following:

public class Test

{

    public static void main(String[] args)

    {

        System.out.print("Y" + "O");

        System.out.print('L');

        System.out.print('O');

    }

}

**Output:** YOLO

**Explanation:** This will now print “YOLO” instead of “YO7679”. It is because the widening primitive conversion happens only when a operator like ‘+’ is present which expects at least integer on both side.

Widening primitive conversion is applied to convert either or both operands as specified by the following rules. The result of adding Java chars, shorts or bytes is an **int**:

* If either operand is of type double, the other is converted to double.
* Otherwise, if either operand is of type float, the other is converted to float.
* Otherwise, if either operand is of type long, the other is converted to long.
* Otherwise, **both operands are converted to type int**

**Type conversion:**

Changing a value from one data type to another type is known as data type conversion. Data type conversions are either widening or narrowing, it depends on the data capacities of the data types involved. There are different ways of, implicitly or explicitly, changing an entity of one data type into another data type.

An important consideration with a type conversion is whether the result of the conversion is within the range of the destination data type.

**Widening Conversion**

If a value of narrower (lower size) data type converted to a value of a broader (higher size) data type without loss of information, is called Widening conversion. This is done implicitly by the JVM and also known as implicit casting. For example an integer data type is directly converted to a double.

### int a = 100;

### double b = a;

### System.out.println(b);

In above example integer (4 Byte) converted into double (8 Byte)

Widening conversion is allowed in the following cases:

* byte can be converted to short, int, long, float, or double
* Short can be converted to int, long, float, or double
* char can be converted to int, long, float, or double
* int can be converted to long, float, or double
* long can be converted to float or double
* float can be converted to double

### Narrowing Conversion :

### If a value of broader (higher size) data type converted to a value of a narrower (lower size) data type which can result in loss of information, is called Narrowing conversion. This is not done implicitly by the JVM and requires explicit casting. For example double data type explicitly converted into integer.

### double a = 100.7;

### int b = (int) a;

### System.out.println(b);

In above example the double 'a' is explicitly converted to int 'b'.

Narrowing conversion is allowed in these cases:

* short can be converted to byte or char
* char can be converted to byte or short
* int can be converted to byte, short, or char
* long can be converted to byte, short, or char
* float can be converted to byte, short, char, int, or long
* double can be converted to byte, short, char, int, long, or float

Narrowing conversion should follow the following thumb rule

* The narrowing conversion must be explicit.
* The target type need to be specified in parentheses.

**Listing 1**: Representing the narrowing conversion

### public class sree {

###  public static void main(String[] args) {

###  long a = 10;

###  int b = (int) a; // narrowing conversion

###  System.out.println(a);

###  System.out.println(b);

###  }

### }

As, in narrowing conversion it may occur information loss, if the converted value is larger than the capacity of the target type. The following example illustrates this.

**Listing 2**: Representing the narrowing conversion

### public class Sree {

###  public static void main(String[] args) {

###  long a = 12345678965;

###  int b = (int) a;

###  System.out.println(b);

###  }

### }

### Automatic conversion :

Automatic type conversion is similar to widening conversion, it occurs implicitly between the compatible types.

An automatic type conversion will occur if the following two conditions are met:

* The two types are compatible.
* The destination type is larger than the source type.

### Type Compatibility :

Two Types are compatible if values of one type can appear wherever values of the other type are expected, and vice versa.

Java is a strongly typed language and from there comes it’s safety and robustness.

Every variable has a type which is strictly defined.

All assignments, whether explicit or via parameter passing in method calls ,are checked for type compatibility and any mismatches result in errors .

### Casting of Incompatible Types :

To perform a conversion between two incompatible types, we must use a cast. A cast is simply an explicit type conversion. It has following general form: (target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer’s value is larger

int a = 257;

int b = (int) a;

**Listing 3**: Representing the casting of incompatible types

public class Sree {

 public static void main(String args[]) {

 byte b;

 int i = 399;

 double d = 424.142;

 System.out.println("\nConversion of int to byte.");

 b = (byte) i;

 System.out.println("i and b " + i + " " + b);

 System.out.println("\nConversion of double to int.");

 i = (int) d;

 System.out.println("d and i " + d + " " + i);

 System.out.println("\nConversion of double to byte.");

 b = (byte) d;

 System.out.println("d and b " + d + " " + b);

 }

}

### Some Common misunderstandings

Implicit and explicit typecasting of primitive types is frequently misunderstood. Below is a small test program, overloaded with multiple constructors to show when and how type conversion promotion is done with implicit casting.

In this example, since the third value is an integer, but there is no constructor that takes an integer, its value and the float value automatically convert to double. (Case 5: double, double, double)

**Listing 4**: Representing the implicit casting

### public class typetest {

### typetest(double a, double b, short c){

### System.out.println("1 (dbldbl short)");

###  }

### typetest(float a, byte b, long c) {

### System.out.println("2 (float byte long)");

###  }

### typetest(long a, long b, long c) {

### System.out.println("3 (long long long)");

###  }

### typetest(float a, long b, short c) {

### System.out.println("4 (float long short)");

###  }

### typetest(double a, double b, double c) {

### System.out.println("5 (dbldbldbl)");

###  }

### public static void main(String[] args) {

### typetest t = new typetest();

### t.typetest(3.4, 3L, 3);

 }

}

**Data Types In Java:**

Before we using a **variable,** we should specify what type (datatype) of variable it is. Because, when we specify the datatype, the system can understand the memory requirements and the operations allowed on the corresponding variables. Suppose we want to store age of a person, then we specify the variable as an integer. Guess we want to store temperature of a city, then we define the it as double.

In Java, an integer **occupies 4 bytes of memory** where as a double occupy 8 bytes of memory.

We can apply almost all operators on integer type data, but we cannot use some operators (like bitwise operators) to double type data. Below are the different types of primitive data types in java with examples listed.

## ****Different Primitive Data Types In Java****

In Java, we have eight basic  (primitive datatypes).

**Each primitive type has its significance**. Out of this eight primitive data types in Java , we have four datatypes to work with integer numbers, two datatypes to work with real numbers (numbers with fraction part), one datatype for character data, and one datatype for logical (boolean) data.

### ****1) BYTE DATA TYPE IN JAVA****

As it occupies 1 byte of memory, we can store a value between **-128 to 127.** If we try to store a value bigger than that we will get a compilation error.

**Eg:**

* byte b=25; is valid
* Byte c=225; is not valid **✕**

To work with single characters (ASCII values up to 127), we can use byte type as it uses only 1 byte of memory (against char which takes 2 bytes of memory).

#### ****2) SHORT DATATYPE IN JAVA:****

It is used to store integers in the range **-32768 to 32767**. Any value out of this range cannot be kept as short. In that case, we should use int. (In C, short is a data modifier but in Java short is a datatype).

##### **3) INT DATA TYPE IN JAVA:**

This is the default integer type. Most of the times we use int type only to work with whole numbers even though they can be managed with byte or short.

We can use up to a 10 digit number **(up to 200 crores nearly)** with int type.

**4) LONG DATATYPE IN JAVA:**

When we want to store a value bigger than int range, we should use long type. With long, we can store up to a 19 digit number. (in C, long is a data modifier but in Java long is a data type).

When using a constant bigger than int range, we should suffix it with ‘l’ or ‘L’ to indicate it to be a long value.

**Eg:**        long a=131009; is fine

* long b=123456789012345; is not fine **✕**
* long c=123456789012345L; is fine

**5) CHAR DATA TYPE IN JAVA:**

In Java, **char type takes 2 bytes of memory** to support UniCode characters ( In C, char type is 1 byte as it supports only ASCII characters). As per UniCode, we have ***” 65536 (216) “*** characters numbered from 0 to 65535. The char type cannot take negative values.

Numeric type can take both positive and negative values. The first **256 (numbered from 0 to 255)** characters of UniCode are the ASCII set of characters only. So UniCode is compatible with ASCII. So both in ASCII and UniCode **‘A’ is 65, ‘B’ is 66, ‘C’ is 67, ‘ a’ is 97, ‘b’ is 98, ‘0’ is 48, ‘1’ is 49, ‘\n’ is 10, ‘\t’ is 9.**

**Eg:**          char a=55; is valid

* char b=’A’; is valid
* char c=-35; is not valid **✕**

**6) FLOAT DATATYPE IN JAVA**

To work with numbers with a **fractional part, we can use float type.** Even though we have float and double types to work with a fractional (decimal) type of data, double is preferred over the float.

Because accuracy with double is better than float. Allowed range of values is more in double than in float. The float cannot take a decimal value directly.

It should be converted to float before assignment. So, an almost float is not used in regular programming.

**Eg:** float a=3.6; is not valid**✕**

* float b=3.6f; is valid
* float c=(float)3.6; is valid

**7) DOUBLE DATA TYPE IN JAVA:**

This is the default datatype to store decimal (real numbers) values. By default, any value with a fractional part is treated as double by the system.

A bigger type (double) value can’t be given to a smaller type (float, int, byte, etc) variable. So the following assignments will result in compilation error.

**Eg: float a=5.8;**

* int b=5.8;
* short c=5.8;

A smaller type value can be given to bigger type variable. So the following statements are valid.

**Eg: double a=3.9;**

* double b=834;
* double c=’A’;

###### **8) Boolean Datatype In Java :**

This is the type used to **store logical values.** In Java, **‘true’ and ‘false’** are reserved words to represent logical (boolean) values.

These are not compatible with other datatypes. So boolean values cannot be assigned to other types and another type of values cannot be assigned to boolean variables.

In java a logical expression results in a logical value (true or false).

**Eg:**

* boolean a=true; is valid
* boolean b=’true’; is not valid **✕**
* boolean c=”false”; is not valid **✕**
* boolean d=false; is valid
* boolean e=1; is not valid **✕**
* boolean f=34; is not valid **✕**
* boolean g=10<20; is valid

**Why Most Of The Programmers Prefer ” Double ” And ” Int ” ?**

Generally, when an operation is performed on bytes or shorts they are internally upcasted to int before the actual operation takes place. Similarly, when we try to add two characters, ***their ASCII values (integers)*** are added. **” Java is more strict in terms of type checking “**.

It does not allow **bigger type data** to be assigned to smaller type. So it is advised to use integer type of even to store small values.

Similarly, when we perform an operation on 2 float type values, they are upcasted to double internally, and the actual operation will be conducted.

By default, the system considers a fractional value as double.

If we try to assign a fraction value to a float variable we may get a compilation error. So it is preferred to use double variables rather than float.

**Reference Environment Variables**

Along with the primitive types, we have a special group of variables known as**reference variables**. These reference variables can refer **objects in the program.**

**Java Variables**

**Java variables** are nothing but a similar way we use a variable in **mathematics** also. Assume if we want to find an [area of a rectangle](http://www.javaprogramshub.com/java-program-area-of-rectangle/), the formula **we use is a=l\*b**. In this expression, **‘a’, ‘l’ and ‘b’ are Java variables**. Usage is same both in**mathematics and programming**.

Variables In Java – Different Types

**How To Create a Variable ? & Different Types**

* **Int a;** < here ‘int’ is datatype and ‘a’ is variable
* **double b;** <  here ‘double’ is a datatype, and ‘b’ is variable
* **char gender;** <  here ‘char’ is datatype and ‘gender’ is variable
* **boolean f1,f2,f3;** <  here ‘boolean’ is datatype and ‘f1’, ‘f2’ and ‘f3’ are variables

**Example – 1: Declaring in java**

Class sree

{

   public static void main(String args[])

    {

        byte b;

        short s;

 int i;

        float f;

        char c;

        double d;

        boolean boo;

        long l;

        System.out.println("Examples created succesfully");

   }

}

**Output:**

Ouput

Java

|  |  |
| --- | --- |
| 1 | Examples created successfully |

**Example -2 : Sorting data :**

Class example

{

   public static void main(String args[])

    {

 int i=5;

        float f=5.5f;

        char c=  'a';

        double d=123456.789;

        boolean b=true;

        System.out.println("Data stored in path successfully");

  }

 }

**Output:**

|  |  |
| --- | --- |
| 1 | Data stored in path successfully |

**Example 2: Declaring Multiple environment Variables**

|  |
| --- |
| Class example{   public static void main(String args[])     {               int i1;  int i2;   int i3;   int i4;     // Or int i1,if,i3,i4;..              float f1;float f2;float f3;float f4;    // Or  float f1,f2,f3,f4;.....              char c1;char c2;char c3;char c4;    // Or char c1,c2,c3,c4;.....                   double d1; double d2; double d3; double d4;    // Or double d1,d2,d3,d4; ....                 boolean b1;boolean b2; boolean b3;boolean b4;    // boolean b1,b2,b3,b4 ....                  System.out.println("more than one example created succesfully");    }} |

**Output:**

Output :

Java

|  |  |
| --- | --- |
| 1 | more than one example created succesfully |

**4)** Keywords cannot be used as environment variables.

Class Example

{

   public static void main(String args[])

    {

         int char;          //  error becuase char is keyword

         int class;        //   error becuase class is keyword

         int int;         //    error becuase int is keyword

         int if;         //     error becuase if is keyword

         int while ;    //      error becuase while is keyword

         int CHAR; // ok becuase Char it is not a keyword because keywords are in " lowercase "

    }

}

**5) When we create a variable/Declaring in a method, we should not create another variable with the same name (even in an inner block) until the first goes out of scope.**

**Example # 5:**

Class Example

{

   public static void main(String args[])

    {

        int a;

 float a;  //error: a is already defined in method main(String[])

        int b;

        int b;   //error: b is already defined in method main(String[])

        float b;  //error: b is already defined in method main(String[])

}

}

**6)** We can create a variable anywhere in the program (in C, an environment variable should be declared at beginning of a block only)

**Example # 6:**

|  |
| --- |
| Class Example{   public static void main(String args[])     {             int a=10;           System.out.println(a);            int b=20;         System.out.println(b);                 int c=a+b;         System.out.println(c);                 int d=40;                System.out.println(d);                int x=22;              } } |

**Output:**

Output :

|  |  |
| --- | --- |
| 1 | 20                                                                                                                                                                                                30                                                                                                                                                                                                  40 |

**7)** Environment Variables can be created at initialization part of a for **loop.** Such are destroyed automatically when the loop ends.

**Example # 7:**

**Class Example**

{

   public static void main(String args[])

    {

 for(int i=1;i<4;i++)

                {

                   System.out.println(i);// 1 2 3

                }

                  System.out.println(i);//error because variable "i"  is vanished after forloop

    }

}

**Java Arrays:**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars;

### Types of Array in java

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

**Single Dimensional Array :**

**Syntax to Declare an Array in Java**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

**Instantiation of an Array in Java**

1. arrayRefVar=**new** datatype[size];

### Example of Java Array

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

1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. **class** Testarray{
4. **public** **static** **void** main(String args[]){
5. **int** a[]=**new** **int**[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }
15. }

Output:

10

20

70

40

50

**Declaration, Instantiation and Initialization of Java Array :**

We can declare, instantiate and initialize the java array together by:

1. **int** a[]={33,3,4,5};//declaration, instantiation and initialization

**Let's see the simple example to print this array.**

1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. **class** Testarray1{
4. **public** **static** **void** main(String args[]){
5. **int** a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. **for**(**int** i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}

**Output:**

1. 33
2. 3
3. 4
4. 5

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

## Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

**Example**

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo

**Note**: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

**Change an Array Element**

To change the value of a specific element, refer to the index number:

Example

cars[0] = "Opel";
**Example**

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo

## Array Length

To find out how many elements an array have, use the length property:

**Example**

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4

### Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.

## Java Array

## Loop Through an Array :

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}

## Loop Through an Array with For-Each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays:

Syntax

for (type variable : arrayname) {
  ...
}

he following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}

The example above can be read like this: **for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of**i**.

If you compare the for loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

## Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

### Example :

Here is a complete example showing how to create, initialize, and process arrays −

public class TestArray {

 public static void main(String[] args) {

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

 // Print all the array elements

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

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

 }

 // Summing all elements

 double total = 0;

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

 total += myList[i];

 }

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

 // Finding the largest element

 double max = myList[0];

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

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

 }

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

 }

}

This will produce the following result −

### Output

### 1.9

### 2.9

### 3.4

### 3.5

### Total is 11.7

### Max is 3.5

## The foreach Loops :

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

### Example

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

### public class TestArray {

###  public static void main(String[] args) {

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

###  // Print all the array elements

###  for (double element: myList) {

###  System.out.println(element);

###  }

###  }

### }

This will produce the following result −

### Output

### 1.9

### 2.9

### 3.4

### 3.5

## Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

Example

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

**myNumbers** is now an array with two arrays as its elements.

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

**Example**

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x);

// Outputs 7

We can also use a for loop inside another for loop to get the elements of a two-dimensional array (we still have to point to the two indexes):

**Example**

public class MyClass {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}

**UNIT- II**

**Java Operators**

## Java Operators

Operators are used to perform operations on variables and values.

The value is called an operand, while the operation (to be performed between the two operands) is defined by an **operator**:

|  |  |  |
| --- | --- | --- |
| **Operand** | **Operator** | **Operand** |
| 100 | + | 50 |

In the example below, the numbers 100 and 50 are **operands**, and the + sign is an **operator**:

Example

int x = 100 + 50;

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and a variable:

Example

int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 =sum1+ 250;      // 400 (150 + 250)
int sum= sum2 + sum2;  // 800 (400 + 400)

Java divides the operators into the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

## Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |  |
| + | Addition | Adds together two values | x + y |  |
| - | Subtraction | Subtracts one value from another | x - y |  |
| \* | Multiplication | Multiplies two values | x \* y |  |
| / | Division | Divides one value from another | x / y |  |
| % | Modulus | Returns the division remainder | x % y |  |
| ++ | Increment | Increases the value of a variable by 1 | ++x |  |
| -- | Decrement | Decreases the value of a variable by 1 | --x |  |

## Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called x:

**Example**

int x = 10;

The **addition assignment** operator (+=) adds a value to a variable:

Example

int x = 10;
x += 5;

A list of all assignment operators:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Same As** |  |
| = | x = 5 | x = 5 |  |
| += | x += 3 | x = x + 3 |  |
| -= | x -= 3 | x = x - 3 |  |
| \*= | x \*= 3 | x = x \* 3 |  |
| /= | x /= 3 | x = x / 3 |  |
| %= | x %= 3 | x = x % 3 |  |
| &= | x &= 3 | x = x & 3 |  |
| |= | x |= 3 | x = x | 3 |  |
| ^= | x ^= 3 | x = x ^ 3 |  |
| >>= | x >>= 3 | x = x >> 3 |  |
| <<= | x <<= 3 | x = x << 3 |  |

## Java Comparison Operators

Comparison operators are used to compare two values:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** |  |
| == | Equal to | x == y |  |
| != | Not equal | x != y |  |
| > | Greater than | x > y |  |
| < | Less than | x < y |  |
| >= | Greater than or equal to | x >= y |  |
| <= | Less than or equal to | x <= y |  |

## Java Logical Operators

Logical operators are used to determine the logic between variables or values:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |  |
| &&  | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 |  |
| ||  | Logical or | Returns true if one of the statements is true | x < 5 || x < 4 |  |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |  |

## The Bitwise Operators :

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

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

a = 0011 1100

b = 0000 1101

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

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators −

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

[Show Examples](https://www.tutorialspoint.com/java/java_bitwise_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & (bitwise and) | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | (bitwise or) | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ (bitwise XOR) | 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 |
| ~ (bitwise compliment) | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << (left shift) | 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 |
| >> (right shift) | 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 |
| >>> (zero fill right shift) | 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 |

**Examples on Operators:**

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

public class Test {

 public static void main(String args[]) {

 boolean a = true;

 boolean b = false;

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

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

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

 }

}

This will produce the following result −

Output

a && b = false

a || b = true

!(a && b) = true

**Example # 1 : Arithmetic**

Class Arithematic

{

    public static void main(String args[])

    {

        int a=1+1;

        int b=a\*2;

 int c=b/2;

        int d=c-1;

        int e=-d;

         System.out.println(a);    //2

         System.out.println(b);    //4

 System.out.println(c);   //2

 System.out.println(d);   //1

   System.out.println(e);   //-1

    }

}

**Example # 2 : Arithmetic Operators In Java**

Class Arithematic

{

    public static void main(String args[])

    {

         double a=1+1;

         double b=a\*2;

 double c=b/2;

         double d=c-1;

         double e=-d;

         System.out.println(a);    //2.0

         System.out.println(b);    //4.0

 System.out.println(c);   //2.0

 System.out.println(d);   //1.0

 System.out.println(e);   //-1.0

    }

}

**Example # 3 : Arithmetic Operators In Java**

Class Arithematic

{

    public static void main(String args[])

    {

         int a=53;

         double b=53.5;

 System.out.println(a%10); //3

         System.out.println(b%10);  //3.5

    }

}

**Example #1 Program For Relational operators in Java :**

Class Relational

{

    public static void main(String args[])

    {

         System.out.println(13>10);    //true

 System.out.println(13<=10);   // true

         System.out.println(13>=10);   // true

 System.out.println(13==10);   // false

 System.out.println(13!=10);   // true

         System.out.println(13>13);    // false

 System.out.println(13>=13);  // true

 System.out.println(13<=13);  // true

         System.out.println(13==13);  //true

 System.out.println(13!=13);  //false

 System.out.println('A'==65); // true    ascii value of'A'is  65

         System.out.println('0'==48); // true    ascii value of'0'is  48

         System.out.println('2'==2 ); // false   ascii value of'2'is  50

         }

}

**Example Programs # 1: Logical Java operators**

Class Logical

{

    public static void main(String args[])

    {

         System.out.println(true  || true);    //true

         System.out.println(true  || false);   // true

         System.out.println(false || true);    // true

         System.out.println(false || false); // false

 System.out.println(true  && true); // true

 System.out.println(true  && false);`   // false

         System.out.println(false && true);     //false

 System.out.println(false && false);    // false

 System.out.println(!true);             // false

         System.out.println(!false);            // true

 System.out.println(13>10 || 13<10);     // true  (true || false)

 System.out.println(13>10 && 13<10);     // false  (true && false)

         System.out.println(!13>10  );           //error

         System.out.println( !(13==10) );        // true   !(false)

    }

}

**Example Programs # Assignment Operators**

Class Assignment

{

    public static void main(String args[])

    {

         int a=2;

         a=a+4;

         System.out.println(a);  //6

         a+=4;

 System.out.println(a);  //10

 a=16;

         a=a-4;

 System.out.println(a);  //12

         a=16;

 a-=4;

 System.out.println(a);  //12

 a=26;

         a=a\*4;

 System.out.println(a);  //104

         a=26;

 a\*=4;

 System.out.println(a);  //104

 a=36;

         a=a/4;

 System.out.println(a);  //9

 a=36;

         a/=4;

 System.out.println(a);  //9

 a=46;

         a=a%4;

 System.out.println(a);  //2

 a=46;

         a%=4;

 System.out.println(a);  //2

      }

    }

**Control statements:**

The control statements, as the name suggests controls the flow of the program. Depending upon the state ( values of variables, user input etc ) the appropriate code is executed. These help in executing or by passing certain parts of the code, or executing them multiple times or skipping some lines of code etc.,.

These control statements in Java can be used together and they can be nested in one other. A program might contain multiple control statements, having them used in the same blocks of code or they can be nested. There is no limitation on the usage of these statements. The control statements in java are of following types :

* [Selection Statements: In Java](http://java.meritcampus.com/core-java-topics/selection-statements-in-java) help in executing or by passing the code blocks. The supported statements are : if-else and switch
* [Iteration Statements or Loops in Java](http://java.meritcampus.com/core-java-topics/iteration-statements-or-loops-in-java) help in executing a block multiple times. The supported statements are : while, do-while and for
* [Jump Statements : In Java](http://java.meritcampus.com/core-java-topics/jump-statements-in-core-java-programming) help in jumping or skipping certain parts of code. The supported statements are : break, continue and return

[**Selection Statements**](http://java.meritcampus.com/core-java-topics?g=501&Selection-statements): Java supports two selection statements - if and switch. These statements allow us to control the flow of the program. Depending upon the expressions or values, the corresponding blocks/statements of code will be executed or by passed.



The two statements are :

* [if](http://java.meritcampus.com/core-java-topics/if-condition-in-java) statement is a conditional branch statement. This is a two way branch statement. Depending upon the whether a condition is true or false, the corresponding code is executed.
* [switch](http://java.meritcampus.com/core-java-topics/switch-statement-in-java) statement is a multiway branch statement. Depending upon the value used for switching, the corresponding code is executed.

 These two statements are very powerful and are used widely across any application. They provide effective solutions for branching problems.

if condition in Java is a conditional branch statement, which can be used to route program execution through different paths.

if condition in Java can be used to execute a block of code only when a condition is true, if the condition is false we can execute a different block of code.

 ***if(condition)
    {
        statement1;
    }
    else
    {
        statement2;
    }***

As shown above, if the condition is true then, *statement1* is executed. If it is false, *statement2* is executed.

class PrintPassFail
{
    public static void main(String arg[])
    {
        int marks = 62;
        if(marks > 35)            // LINE A
        {
            System.out.println("Pass");     // LINE B
        }
        else
        {
            System.out.println("Fail");     // LINE C
        }

    }
}

**OUTPUT**

Pass

**DESCRIPTION**

Here marks are initialized to 62. Then the condition marks > 35 will become true, so LINE B is executed which prints Pass. If the marks are only 20, instead of 62, then the condition is false, causing LINE C to execute, printing Fail.

**THINGS TO TRY**

* Try the below code.

int a = 10;
int b = 5;
if (a > b)
    System.out.println("a is greater than b");
else
    System.out.println("b is greate than a");

The output should be a is greater than b, since the value of a is greater than b.

* Try the below code.

int a = 10;
int b = 9;
if (a > b)
System.out.println("a is greater than b");
System.out.println("I am not in if block");

The output should be as shown.
a is greater than b
I am not in if block
Only the first statement below if condition comes into if block.

When if condition is included in the if block of some other if condition then those are called nested ifs.

 We can have as many nested ifs and it can go into many levels.

class ClassifyPerson
{
    public static void main(String arg[])
    {
        int age = 35;
        char gender = 'F'; // M - Male, F - Female

        if( age > 35 ) // outer if
        {
            if( gender == 'M' )    // LINE A
            {
                System.out.println("Man");
            }
            else
            {
                System.out.println("Woman");
            }
        }
        else
        {
            if( gender == 'M' )    // LINE B
            {
                System.out.println("Boy");
            }
            else
            {
                System.out.println("Girl");
            }

        }

    }
}

**OUTPUT**

Girl

**DESCRIPTION**

Here we have two variables age and gender. In the outer if the condition age > 35 is checked, if its true, the block starting with LINE A is executed. The inner if condition present in LINE A checks for gender and prints Man or Woman. If age is less than 35 then the else block starting with LINE B is executed. The if in LINE B checks for gender and prints Boy or Girl.

**THINGS TO TRY**

Add one more variable marks of type int to the above program and modify the program such that it prints Intelligent when marks greater than or equal to 75, otherwise it prints Dull. and If age greater than or equal to 35 and gender equal to M it prints Man otherwise it prints Woman. Boy or Girl when age less than 35.
Examples:
1. when marks are 75, age is 35 and gender is F output should be Intelligent Woman
2. marks are 40, age is 22 and gender is M output should be Dull Boy

It is very common in programming to have more than two ways (or branches). e.g., printing the grade of a student like "Distinction", "First Class", "Second Class" and "Fail" instead of simply "Pass" or "Fail". In this case if-else-if ladder will be very useful.

***if( condition1 )
    {
        statement1; // BLOCK 1
    }
    else if( condition2 )
    {
        statement2; // BLOCK 2
    }
    else if( condition3 )
    {
        statement3; // BLOCK 3
    }
    else
    {
        statement4; // BLOCK 4
    }***

The above if conditions are executed top down. If the *condition1* is true, only *statement1* is executed, the other statements - *statement2* and *statement3* will not be executed.
If the *condition1* is false and *condition2* is true, then only *statement2* is executed. If *condition1*, *condition2* are false but *condition3* is true, then only *statement3* is executed. If all the conditions - *condition1*, *condition2* and *condition3* are false, then *statement4* is executed.
Also note that when *condition1* is true, irrespective of whether other conditions are true or false, only *statement1* will be executed. *statement2* and *statement3* will not be executed.

class PrintStudentGrade
{
    public static void main(String arg[])
    {
        int marks = 65;

        if( marks > 75 )      // CONDITION A
        {
            System.out.println("Distinction"); // LINE A
        }
        else if( marks > 60 ) // CONDITION B
        {
            System.out.println("First Class"); // LINE B
        }
        else if( marks > 50 ) // CONDITION C
        {
            System.out.println("Second Class"); // LINE C
        }
        else
        {
            System.out.println("Fail");  // LINE D
        }

    }
}

OUTPUT

First Class

**UNIT- III**

**Super Keyword in Java**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

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

**Usage of Java super Keyword**

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



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

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

1. **class** Animal{
2. String color="white";
3. }
4. **class** Dog **extends** Animal{
5. String color="black";
6. **void** printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(**super**.color);//prints color of Animal class
9. }
10. }
11. **class** TestSuper1{
12. **public** **static** **void** main(String args[]){
13. Dog d=**new** Dog();
14. d.printColor();
15. }}

**Output:**

1. black
2. white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

**2) 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 if subclass contains the same method as parent class. In other words, it is used if method is overridden.

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. **void** bark(){System.out.println("barking...");}
7. **void** work(){
8. **super**.eat();
9. bark();
10. }
11. }
12. **class** TestSuper2{
13. **public** **static** **void** main(String args[]){
14. Dog d=**new** Dog();
15. d.work();
16. }}

Output:

1. eating...
2. barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

**3) super is used to invoke parent class constructor.**

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. **super**();
7. System.out.println("dog is created");
8. }
9. }
10. **class** TestSuper3{
11. **public** **static** **void** main(String args[]){
12. Dog d=**new** Dog();
13. }}

**Output:**

1. animal is created
2. dog is created

#### Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

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

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. **class** TestSuper4{
10. **public** **static** **void** main(String args[]){
11. Dog d=**new** Dog();
12. }}

Output:

1. animal is created
2. dog is created

[**Multilevel Hierarchy in java programming**](http://codercheck.blogspot.com/2012/11/multilevel-hierarchy-in-java-programming.html)

In simple inheritance a subclass or derived class derives the properties from its parent class, but in multilevel inheritance a subclass is derived from a derived class. One class inherits only single class. Therefore, in multilevel inheritance, every time ladder increases by one. The lower most class will have the properties of all the super classes’.

It is common that a class is derived from another derived class. The class student serves as a base class for the derived class marks, which in turn serves as a base class for the derived class percentage. The class marks is known as intermediates base class since it provides a link for the inheritance between student and percentage.

The chain is known as inheritance path. When this type of situation occurs, each subclass inherits all of the features found in all of its super classes. In this case, percentage inherits all aspects of marks and student.

 **To understand the flow of program read all comments of program.**

class student
{
    int rollno;
    String name;

    student(int r, String n)
    {
        rollno = r;
        name = n;
    }
    void dispdatas()
    {
        System.out.println("Rollno = " + rollno);
        System.out.println("Name = " + name);
    }
}

class marks extends student
{
    int total;
    marks(int r, String n, int t)
    {
        super(r,n);   //call super class (student) constructor
        total = t;
    }
    void dispdatam()
    {
        dispdatas();    // call dispdatap of student class
        System.out.println("Total = " + total);
    }
}

class percentage extends marks
{
    int per;

    percentage(int r, String n, int t, int p)
    {
        super(r,n,t);  //call super class(marks) constructor
        per = p;
    }
    void dispdatap()
    {
        dispdatam();    // call dispdatap of marks class
        System.out.println("Percentage = " + per);
    }
}
class Multi\_Inhe
{
    public static void main(String args[])
    {
        percentage stu = new percentage(102689, "Srilakshmi", 350, 70); //call constructor percentage
        stu.dispdatap();  // call dispdatap of percentage class
    }
}

**Output**

G:\>javac Multi\_Inhe.java
G:\>java Multi\_Inhe
Rollno = 102689
Name = srilakshmi
Total = 350
Percentage = 70

#  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 a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an 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. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. class Vehicle{
6. void run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. class Bike extends Vehicle{
10. public static void main(String args[]){
11. //creating an instance of child class
12. Bike obj = new Bike();
13. //calling the method with child class instance
14. obj.run();
15. }
16. }
17. Output:
18. 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 are the same, and there is IS-A relationship between the classes, so there is method overriding.

1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. **class** Vehicle{
4. //defining a method
5. **void** run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. **class** Bike2 **extends** Vehicle{
9. //defining the same method as in the parent class
10. **void** run(){System.out.println("Bike is running safely");}
11.
12. **public** **static** **void** main(String args[]){
13. Bike2 obj = **new** Bike2();//creating object
14. obj.run();//calling method
15. }
16. }

**Output:**

Bike is running safely

### A real example of Java Method Overriding

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



### Runtime Polymorphism or Dynamic method dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



#### Upcasting :

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

#### Example

class Game

{

 public void type()

 { System.out.println("Indoor & outdoor"); }

}

Class Cricket extends Game

{

 public void type()

 { System.out.println("outdoor game"); }

 public static void main(String[] args)

 {

 Game gm = new Game();

 Cricket ck = new Cricket();

 gm.type();

 ck.type();

 gm=ck; //gm refers to Cricket object

 gm.type(); //calls Cricket's version of type

 }

}

Output:

Indoor & outdoor

Outdoor game

Outdoor game

#### Difference between Static binding and Dynamic binding in java ?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

In simpler terms, Static binding means when the type of object which is invoking the method is determined at compile time by the compiler. While Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler.

**Abstract class in Java :**

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the 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 essential things to the user and hides the internal details, for example, sending SMS where you 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 Abstraction :

There are two ways to achieve abstraction in java

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

### Abstract class in Java :

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### Points to Remember :

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.



**Example of abstract class :**

**abstract** **class** A{}

### Abstract Method in Java :

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

**Example of abstract method :**

**abstract** **void** printStatus();//no method body and abstract

### Example of Abstract class that has an abstract method

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

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

O/p:

running safely

Using final with Inheritance in Java

[**final**](https://www.geeksforgeeks.org/g-fact-48/) is a keyword in java used for restricting some functionalities. We can declare variables, methods and classes with final keyword.

**Using final with inheritance**

During inheritance, we must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. Note that it is not necessary to declare final methods in the initial stage of inheritance(base class always). We can declare final method in any subclass for which we want that if any other class extends this subclass, then it must follow same implementation of the method as in the that subclass.

Java program to illustrate

// use of final with inheritance

// base class

abstract class Shape

{

    private double width;

    private double height;

    // Shape class parameterized constructor

    public Shape(double width, double height)

    {

        this.width = width;

        this.height = height;

    }

    // getWidth method is declared as final

    // so any class extending

    // Shape cann't override it

    public final double getWidth()

    {

        return width;

    }

    // getHeight method is declared as final

    // so any class extending Shape

    // can not override it

    public final double getHeight()

    {

        return height;

    }

// method getArea() declared abstract because

    // it upon its subclasses to provide

    // complete implementation

    abstract double getArea();

}

// derived class one

class Rectangle extends Shape

{

    // Rectangle class parameterized constructor

    public Rectangle(double width, double height)

    {

        // calling Shape class constructor

        super(width, height);

    }

    // getArea method is overridden and declared

    // as final so any class extending

    // Rectangle cann't override it

    @Override

    final double getArea()

    {

        return this.getHeight() \* this.getWidth();

    }

}

//derived class two

class Square extends Shape

{

    // Rectangle class parameterized constructor

    public Square(double side)

    {

        // calling Shape class constructor

        super(side, side);

    }

// getArea method is overridden and declared as

    // final so any class extending

    // Square cann't override it

    @Override

    final double getArea()

    {

        return this.getHeight() \* this.getWidth();

    }

}

// Driver class

public class Test

{

    public static void main(String[] args)

    {

        // creating Rectangle object

        Shape s1 = new Rectangle(10, 20);

        // creating Square object

        Shape s2 = new Square(10);

        // getting width and height of s1

        System.out.println("width of s1 : "+ s1.getWidth());

        System.out.println("height of s1 : "+ s1.getHeight());

        // getting width and height of s2

        System.out.println("width of s2 : "+ s2.getWidth());

        System.out.println("height of s2 : "+ s2.getHeight());

        //getting area of s1

        System.out.println("area of s1 : "+ s1.getArea());

|  |
| --- |
| //getting area of s2         System.out.println("area of s2 : "+ s2.getArea());               } }  |

Output:

width of s1 : 10.0

height of s1 : 20.0

width of s2 : 10.0

height of s2 : 10.0

area of s1 : 200.0

area of s2 : 100.0

When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it. This is particularly useful, for example, when [creating an immutable class](https://www.geeksforgeeks.org/create-immutable-class-java/) like the predefined [String](http://quiz.geeksforgeeks.org/string-class-in-java/)class. The following fragment illustrates **final** keyword with a class:

final class A

{

 // methods and fields

}

// The following class is illegal.

class B extends A

{

 // ERROR! Can't subclass A

}

**Note :**

* Declaring a class as final implicitly declares all of its methods as final, too.
* It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. For more on abstract classes, refer [abstract classes in java](https://www.geeksforgeeks.org/abstract-classes-in-java/)

Object class in Java :

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

### Methods of Object class

|  |
| --- |
| The Object class provides many methods. They are as follows: |



|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

# Java Package :

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

## Advantage of Java Package :

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



**Simple example of java package**

The **package keyword** is used to create a package in java.

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

**How to compile java package**

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

**How to run java package program**

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |

 Output:Welcome to package

|  |
| --- |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## Example of package that import the packagename.\*

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9.
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. A obj = **new** A();
13. obj.msg();
14. }
15. }

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java
2.
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
8. **package** mypack;
9. **import** pack.A;
10.
11. **class** B{
12. **public** **static** **void** main(String args[]){
13. A obj = **new** A();
14. obj.msg();
15. }
16. }

Output:Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified name
11. obj.msg();
12. }
13. }

Output:Hello

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.



## Subpackage in java :

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### Example of Subpackage

1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java com.javatpoint.core.Simple |

Output:Hello subpackage

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

### To Compile:

**e:\sources> javac -d c:\classes Simple.java**

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| **e:\sources> set classpath=c:\classes;.;** |
| **e:\sources> java mypack.Simple** |

### Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

**e:\sources> java -classpath c:\classes mypack.Simple**

Output: Welcome to package

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
	+ By setting the classpath in the command prompt
	+ By -classpath switch
* Permanent
	+ By setting the classpath in the environment variables
	+ By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

1. //save as C.java otherwise Compilte Time Error
2.
3. **class** A{}
4. **class** B{}
5. **public** **class** C{}

### How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example: |

1. //save as A.java
2.
3. **package** javatpoint;
4. **public** **class** A{}
5. //save as B.java
6.
7. **package** javatpoint;
8. **public** **class** B{}

### What is static import feature of Java5?

|  |
| --- |
| Click [Static Import](https://www.javatpoint.com/static-import-in-java) feature of Java5. |

# Interfaces:

# Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

**Why use Java interface?**

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

## How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

### Syntax:

1. **interface** <interface\_name>{
2.
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

**Java 8 Interface Improvement**

Since Java 8, interface can have default and static methods which is discussed later.

Internal addition by the compiler

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

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



#### The 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**.



**Java Interface Example**

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public** **void** print(){System.out.println("Hello");}
6.
7. **public** **static** **void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

**Output:**

Hello

**Java Interface Example: Drawable**

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

***File: TestInterface1.java***

1. //Interface declaration: by first user
2. **interface** Drawable{
3. **void** draw();
4. }
5. //Implementation: by second user
6. **class** Rectangle **implements** Drawable{
7. **public** **void** draw(){System.out.println("drawing rectangle");}
8. }
9. **class** Circle **implements** Drawable{
10. **public** **void** draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. **class** TestInterface1{
14. **public** **static** **void** main(String args[]){
15. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}

Output: drawing circle

**Multiple inheritance in Java by interface :**

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** show();
6. }
7. **class** A7 **implements** Printable,Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
10.
11. **public** **static** **void** main(String args[]){
12. A7 obj = **new** A7();
13. obj.print();
14. obj.show();
15. }
16. }

Output: Hello

 Welcome

**Java 8 Default Method in Interface**

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

*File: TestInterfaceDefault.java*

1. **interface** Drawable{
2. **void** draw();
3. **default** **void** msg(){System.out.println("default method");}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** TestInterfaceDefault{
9. **public** **static** **void** main(String args[]){
10. Drawable d=**new** Rectangle();
11. d.draw();
12. d.msg();
13. }}

Output:

drawing rectangle

default method

**Java 8 Static Method in Interface**

Since Java 8, we can have static method in interface. Let's see an example:

***File: TestInterfaceStatic.java***

1. **interface** Drawable{
2. **void** draw();
3. **static** **int** cube(**int** x){**return** x\*x\*x;}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8.
9. **class** TestInterfaceStatic{
10. **public** **static** **void** main(String args[]){
11. Drawable d=**new** Rectangle();
12. d.draw();
13. System.out.println(Drawable.cube(3));
14. }}

**Output:**

drawing rectangle

27

**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. Since Java 8, it can have **default and static methods** also. |
| 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 provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class**can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class**can be extended using keyword "extends". | An **interface class** can be implemented using keyword "implements". |
| 8) A Java**abstract class**can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**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%).

Exceptions in Java

**What is an Exception?**

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time that disrupts the normal flow of the program’s instructions.

**Error vs Exception**

**Error:**An Error indicates serious problem that a reasonable application should not try to catch.
**Exception:**Exception indicates conditions that a reasonable application might try to catch.

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.Another branch,**Error** are used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



**How JVM handle an Exception?**

**Default Exception Handling:**Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system (JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

* The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
* The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
* If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
* If run-time system searches all the methods on call stack and couldn’t have found the appropriate handler then run-time system handover the Exception Object to **default exception handler,** which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

Exception in thread "xxx" Name of Exception: Description

... ...... .. // Call Stack

See the below diagram to understand the flow of the call stack.



**Example** :

|  |
| --- |
| // Java program to demonstrate how exception is thrown. class ThrowsExecp{           public static void main(String args[]){           String str = null;         System.out.println(str.length());               } }  |

**Output** :

Exception in thread "main" java.lang.NullPointerException

 at ThrowsExecp.main(File.java:8)

Let us see an example that illustrate how run-time system searches appropriate exception handling code on the call stack:

|  |
| --- |
| ***// Java program to demonstrate exception is thrown*** ***// how the runTime system searches th call stack*** ***// to find appropriate exception handler.*** class ExceptionThrown {     // It throws the Exception(ArithmeticException).     // Appropriate Exception handler is not found within this method.     static int divideByZero(int a, int b){                   // this statement will cause ArithmeticException(/ by zero)         int i = a/b;                    return i;     }           // The runTime System searches the appropriate Exception handler     // in this method also but couldn't have found. So looking forward     // on the call stack.     static int computeDivision(int a, int b) {                   int res =0;                   try        {           res = divideByZero(a,b);         }         // doesn't matches with ArithmeticException         catch(NumberFormatException ex)         {            System.out.println("NumberFormatException is occured");          }         return res;     } ***// In this method found appropriate Exception handler.*** ***// i.e. matching catch block.***     public static void main(String args[]){         int a = 1;         int b = 0;          try        {             int i = computeDivision(a,b);                   }                   // matching ArithmeticException         catch(ArithmeticException ex)         {             // getMessage will print description of exception(here / by zero)             System.out.println(ex.getMessage());         }     } }  |

Output :

/ by zero.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

## Catching Exceptions :

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

**Syntax**

try {

 // Protected code

} catch (ExceptionName e1) {

 // Catch block

}

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follow the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

### Example :

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

### // File Name : ExcepTest.java

### import java.io.\*;

### public class ExcepTest {

###  public static void main(String args[]) {

###  try {

###  int a[] = new int[2];

###  System.out.println("Access element three :" + a[3]);

###  } catch (ArrayIndexOutOfBoundsException e) {

###  System.out.println("Exception thrown :" + e);

###  }

###  System.out.println("Out of the block");

###  }

### }

This will produce the following result −

### Output

**Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3**

**Out of the block**

## Multiple Catch Blocks :

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

**Syntax**

try {

 // Protected code

} catch (ExceptionType1 e1) {

 // Catch block

} catch (ExceptionType2 e2) {

 // Catch block

} catch (ExceptionType3 e3) {

 // Catch block

}

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

### Example

Here is code segment showing how to use multiple try/catch statements.

### try {

###  file = new FileInputStream(fileName);

###  x = (byte) file.read();

### } catch (IOException i) {

###  i.printStackTrace();

###  return -1;

### } catch (FileNotFoundException f) // Not valid! {

###  f.printStackTrace();

###  return -1;

### }

## The Throws/Throw Keywords :

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, ***throws***is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

### Example

### import java.io.\*;

### public class className {

###  public void deposit(double amount) throws RemoteException {

###  // Method implementation

###  throw new RemoteException();

###  }

###  // Remainder of class definition

### }

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException −

### Example

### import java.io.\*;

### public class className {

###  public void withdraw(double amount) throws RemoteException,

###  InsufficientFundsException {

###  // Method implementation

###  }

###  // Remainder of class definition

### }

## The Finally Block :

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

### Syntax

## try {

##  // Protected code

## } catch (ExceptionType1 e1) {

##  // Catch block

## } catch (ExceptionType2 e2) {

##  // Catch block

## } catch (ExceptionType3 e3) {

##  // Catch block

## }finally {

##  // The finally block always executes.

## }

### Example

### public class ExcepTest {

###  public static void main(String args[]) {

###  int a[] = new int[2];

###  try {

###  System.out.println("Access element three :" + a[3]);

###  } catch (ArrayIndexOutOfBoundsException e) {

###  System.out.println("Exception thrown :" + e);

###  }finally {

###  a[0] = 6;

###  System.out.println("First element value: " + a[0]);

###  System.out.println("The finally statement is executed");

###  }

###  }

### }

This will produce the following result −

### Output :

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

**Note the following −**

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses whenever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.

**Built in Exception:**

| **Sr.No.** | **Exception & Description** |
| --- | --- |
| 1 | **ArithmeticException**Arithmetic error, such as divide-by-zero. |
| 2 | **ArrayIndexOutOfBoundsException**Array index is out-of-bounds. |
| 3 | **ArrayStoreException**Assignment to an array element of an incompatible type. |
| 4 | **ClassCastException**Invalid cast. |
| 5 | **IllegalArgumentException**The illegal argument used to invoke a method. |
| 6 | **IllegalMonitorStateException**Illegal monitor operation, such as waiting on an unlocked thread. |
| 7 | **IllegalStateException**Environment or application is in an incorrect state. |
| 8 | **IllegalThreadStateException**Requested operation not compatible with the current thread state. |
| 9 | **IndexOutOfBoundsException**Some type of index is out-of-bounds. |
| 10 | **NegativeArraySizeException**The array created with a negative size. |
| 11 | **NullPointerException**Invalid use of a null reference. |
| 12 | **NumberFormatException**Invalid conversion of a string to a numeric format. |
| 13 | **SecurityException**Attempt to violate security. |
| 14 | **StringIndexOutOfBounds**Attempt to index outside the bounds of a string. |
| 15 | **UnsupportedOperationException**An unsupported operation was encountered. |

* Java defines several exception classes inside the standard package java.lang.
* The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.
* Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Following is the list of Java Checked Exceptions Defined in java.lang.

| **Sr.No.**

|  |
| --- |
|  |

 | **Exception & Description** |
| 1 | **ClassNotFoundException**Class not found. |
| 2 | **CloneNotSupportedException**Attempt to clone an object that does not implement the Cloneable interface. |
| 3 | **IllegalAccessException**Access to a class is denied. |
| 4 | **InstantiationException**Attempt to create an object of an abstract class or interface. |
| 5 | **InterruptedException**One thread has been interrupted by another thread. |
| 6 | **NoSuchFieldException**A requested field does not exist. |
| 7 | **NoSuchMethodException**A requested method does not exist. |

**Creating our own exceptions:**

**User-defined Exceptions:**

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −

* All exceptions must be a child of Throwable.
* If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
* If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below −

**class MyException extends Exception {**

**}**

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

### Example

// File Name InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception {

 private double amount;

 public InsufficientFundsException(double amount) {

 this.amount = amount;

 }

 public double getAmount() {

 return amount;

 }

}

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

**// File Name CheckingAccount.java**

import java.io.\*;

public class CheckingAccount {

 private double balance;

 private int number;

 public CheckingAccount(int number) {

 this.number = number;

 }

 public void deposit(double amount) {

 balance += amount;

 }

 public void withdraw(double amount) throws InsufficientFundsException {

 if(amount <= balance) {

 balance -= amount;

 }else {

 double needs = amount - balance;

 throw new InsufficientFundsException(needs);

 }

 }

 public double getBalance() {

 return balance;

 }

 public int getNumber() {

 return number;

 }

}

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

// File Name BankDemo.java

public class BankDemo {

 public static void main(String [] args) {

 CheckingAccount c = new CheckingAccount(101);

 System.out.println("Depositing $500...");

 c.deposit(500.00);

 try {

 System.out.println("\nWithdrawing $100...");

 c.withdraw(100.00);

 System.out.println("\nWithdrawing $600...");

 c.withdraw(600.00);

 } catch (InsufficientFundsException e) {

 System.out.println("Sorry, but you are short $" + e.getAmount());

 e.printStackTrace();

 }

 }

}

Compile all the above three files and run BankDemo. This will produce the following result −

### Output :

Depositing $500...

Withdrawing $100...

Withdrawing $600...

Sorry, but you are short $200.0

InsufficientFundsException

 at CheckingAccount.withdraw(CheckingAccount.java:25)

 at BankDemo.main(BankDemo.java:13)

UNIT – IV

Multithreaded Programming

# [Explain Java Thread Model](http://ecomputernotes.com/java/multithreading/java-thread-model) :

# The Java language and its run-time system was designed keeping in mind about multithreading. The run-time system depend upon multithreading. Java provides asynchronous thread environment, this helps to increase the utilization of CPU.

Multithreading is best in all cases in contrast with single-thread model. Single-thread system uses an approach of event loop with polling. According to this approach a single thread in the system runs in an infinite loop. Polling the mechanism, that selects a single event from the event queue to choose what to do next. As the event is selected, then event loop forwards the control to the corresponding required event handler. Nothing else can be happened, until the event handler returns. Because of this CPU time is wasted. Here, only one part of the complete program is dominating the whole system, and preventing the system to execute or start any other process. In single-thread model one thread blocks all other threads until its execution completes. On other waiting or idle thread can start and acquire the resource which is not in use by the current thread. This causes the wastage of resources.

Java's multithreading provides benefit in this area by eliminating the loop and polling mechanism, one thread can be paused without stopping the other parts of the program. If any thread is paused or blocked, still other threads continue to run.

As the process has several states, similarly a thread exists in several states. A thread can be in the following states:

*Ready to run (New):*First time as soon as it gets CPU time.

*Running:*Under execution.

*Suspended:*Temporarily not active or under execution.

*Blocked:*Waiting for resources.

*Resumed:*Suspended thread resumed, and start from where it left off.

*Terminated:*Halts the execution immediately and never resumes.



**Java thread model can be defined in the following three sections:**

**Thread Priorities**

Each thread has its own priority in Java. Thread priority is an absolute integer value. Thread priority decides only when a thread switches from one running thread to next, called *context switching.*Priority does increase the running time of the thread or gives faster execution.

**Synchronization**

Java supports an asynchronous multithreading, any number of thread can run simultaneously without disturbing other to access individual resources at different instant of time or shareable resources. But some time it may be possible that shareable resources are used by at least two threads or more than two threads, one has to write at the same time, or one has to write and other thread is in the middle of reading it. For such type of situations and circumstances Java implements synchronization model called *monitor.* The monitor was first defined by C.A.R. Hoare. You can consider the monitor as a box, in which only one thread can reside. As a thread enter in monitor, all other threads have to wait until that thread exits from the monitor. In such a way, a monitor protects the shareable resources used by it being manipulated by other waiting threads at the same instant of time. Java provides a simple methodology to implement

synchronization.

**Messaging**

A program is a collection of more than one thread. Threads can communicate with each other. Java supports messaging between the threads with lost-cost. It provides methods to all objects for inter-thread communication. As a thread exits from synchronization state, it notifies all the waiting threads.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### Thread class:

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

### Commonly used Constructors of Thread class:

|  |
| --- |
| * Thread()
* Thread(String name)
* Thread(Runnable r)
* Thread(Runnable r,String name)
 |

### Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread.
 |

### Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:* A new thread starts(with new callstack).
* The thread moves from New state to the Runnable state.
* When the thread gets a chance to execute, its target run() method will run.
 |

### 1) Java Thread Example by extending Thread class

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

Output:thread is running...

### 2) Java Thread Example by implementing Runnable interface

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5.
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

Output:thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

**Thread Example:**

public class SimpleThreads {

 // Display a message, preceded by

 // the name of the current thread

 static void threadMessage(String message) {

 String threadName =

 Thread.currentThread().getName();

 System.out.format("%s: %s%n",

 threadName,

 message);

 }

 private static class MessageLoop

 implements Runnable {

 public void run() {

 String importantInfo[] = {

 "Mares eat oats",

 "Does eat oats",

 "Little lambs eat ivy",

 "A kid will eat ivy too"

 };

 try {

 for (int i = 0;

 i < importantInfo.length;

 i++) {

 // Pause for 4 seconds

 Thread.sleep(4000);

 // Print a message

 threadMessage(importantInfo[i]);

 }

 } catch (InterruptedException e) {

 threadMessage("I wasn't done!");

 }

 }

 }

 public static void main(String args[])

 throws InterruptedException {

 // Delay, in milliseconds before

 // we interrupt MessageLoop

 // thread (default one hour).

 long patience = 1000 \* 60 \* 60;

 // If command line argument

 // present, gives patience

 // in seconds.

 if (args.length > 0) {

 try {

 patience = Long.parseLong(args[0]) \* 1000;

 } catch (NumberFormatException e) {

 System.err.println("Argument must be an integer.");

 System.exit(1);

 }

 }

 threadMessage("Starting MessageLoop thread");

 long startTime = System.currentTimeMillis();

 Thread t = new Thread(new MessageLoop());

 t.start();

 threadMessage("Waiting for MessageLoop thread to finish");

 // loop until MessageLoop

 // thread exits

 while (t.isAlive()) {

 threadMessage("Still waiting...");

 // Wait maximum of 1 second

 // for MessageLoop thread

 // to finish.

 t.join(1000);

 if (((System.currentTimeMillis() - startTime) > patience)

 && t.isAlive()) {

 threadMessage("Tired of waiting!");

 t.interrupt();

 // Shouldn't be long now

 // -- wait indefinitely

 t.join();

 }

 }

 threadMessage("Finally!");

 }

}

**Java Input and output basics:**

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

**Stream :**

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:**standard output stream

**2) System.in:**standard input stream

**3) System.err:**standard error stream

Let's see the code to print **output and an error** message to the console.

1. System.out.println("simple message");
2. System.err.println("error message");

Let's see the code to get **input** from console.

1. **int** i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((**char**)i);//will print the character

## OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

### OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

### InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



**OutputStream class :**

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

### Useful methods of OutputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public void write(int)throws IOException | is used to write a byte to the current output stream. |
| 2) public void write(byte[])throws IOException | is used to write an array of byte to the current output stream. |
| 3) public void flush()throws IOException | flushes the current output stream. |
| 4) public void close()throws IOException | is used to close the current output stream. |

### OutputStream Hierarchy



## InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Useful methods of InputStream



# Java FileWriter Class :

Java FileWriter class is used to write character-oriented data to a [file](https://www.javatpoint.com/java-file-class). It is character-oriented class which is used for file handling in [java](https://www.javatpoint.com/java-tutorial).

Unlike FileOutputStream class, you don't need to convert string into byte [array](https://www.javatpoint.com/array-in-java) because it provides method to write string directly.

## Java FileWriter class declaration

Let's see the declaration for Java.io.FileWriter class:

1. **public** **class** FileWriter **extends** OutputStreamWriter

## Constructors of FileWriter class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| FileWriter(String file) | Creates a new file. It gets file name in [string](https://www.javatpoint.com/java-string). |
| FileWriter(File file) | Creates a new file. It gets file name in File [object](https://www.javatpoint.com/object-and-class-in-java). |

**Methods of FileWriter class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void write(String text) | It is used to write the string into FileWriter. |
| void write(char c) | It is used to write the char into FileWriter. |
| void write(char[] c) | It is used to write char array into FileWriter. |
| void flush() | It is used to flushes the data of FileWriter. |
| void close() | It is used to close the FileWriter. |

**Java FileWriter Example**

In this example, we are writing the data in the file testout.txt using Java FileWriter class.

1. **package** com.javatpoint;
2. **import** java.io.FileWriter;
3. **public** **class** FileWriterExample {
4. **public** **static** **void** main(String args[]){
5. **try**{
6. FileWriter fw=**new** FileWriter("D:\\testout.txt");
7. fw.write("Welcome to javaTpoint.");
8. fw.close();
9. }**catch**(Exception e){System.out.println(e);}
10. System.out.println("Success...");
11. }
12. }

Output:

Success...

testout.txt:

Welcome to javaTpoint.

# Java FileReader Class :

Java FileReader class is used to read data from the file. It returns data in byte format like [FileInputStream](https://www.javatpoint.com/java-fileinputstream-class) class.

It is character-oriented class which is used for [file](https://www.javatpoint.com/java-file-class) handling in [java](https://www.javatpoint.com/java-tutorial).

**Java FileReader class declaration**

Let's see the declaration for Java.io.FileReader class:

1. **public** **class** FileReader **extends** InputStreamReader

## Constructors of FileReader class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| FileReader(String file) | It gets filename in [string](https://www.javatpoint.com/java-string). It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException. |
| FileReader(File file) | It gets filename in [file](https://www.javatpoint.com/java-file-class) instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException. |

**Methods of FileReader class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| int read() | It is used to return a character in ASCII form. It returns -1 at the end of file. |
| void close() | It is used to close the FileReader class. |

**Java FileReader Example**

In this example, we are reading the data from the text file **testout.txt** using Java FileReader class.

1. **package** com.javatpoint;
2.
3. **import** java.io.FileReader;
4. **public** **class** FileReaderExample {
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileReader fr=**new** FileReader("D:\\testout.txt");
7. **int** i;
8. **while**((i=fr.read())!=-1)
9. System.out.print((**char**)i);
10. fr.close();
11. }
12. }

Here, we are assuming that you have following data in "testout.txt" file:

Welcome to javaTpoint.

Output:

Welcome to javaTpoint.