* Java - Quick Guide
* Java Tutorial
* [Java - Home](https://www.tutorialspoint.com/java/index.htm)
* [Java - Overview](https://www.tutorialspoint.com/java/java_overview.htm)
* [Java - Environment Setup](https://www.tutorialspoint.com/java/java_environment_setup.htm)
* [Java - Basic Syntax](https://www.tutorialspoint.com/java/java_basic_syntax.htm)
* [Java - Object & Classes](https://www.tutorialspoint.com/java/java_object_classes.htm)
* [Java - Basic Datatypes](https://www.tutorialspoint.com/java/java_basic_datatypes.htm)
* [Java - Variable Types](https://www.tutorialspoint.com/java/java_variable_types.htm)
* [Java - Modifier Types](https://www.tutorialspoint.com/java/java_modifier_types.htm)
* [Java - Basic Operators](https://www.tutorialspoint.com/java/java_basic_operators.htm)
* [Java - Loop Control](https://www.tutorialspoint.com/java/java_loop_control.htm)
* [Java - Decision Making](https://www.tutorialspoint.com/java/java_decision_making.htm)
* [Java - Numbers](https://www.tutorialspoint.com/java/java_numbers.htm)
* [Java - Characters](https://www.tutorialspoint.com/java/java_characters.htm)
* [Java - Strings](https://www.tutorialspoint.com/java/java_strings.htm)
* [Java - Arrays](https://www.tutorialspoint.com/java/java_arrays.htm)
* [Java - Date & Time](https://www.tutorialspoint.com/java/java_date_time.htm)
* [Java - Regular Expressions](https://www.tutorialspoint.com/java/java_regular_expressions.htm)
* [Java - Methods](https://www.tutorialspoint.com/java/java_methods.htm)
* [Java - Files and I/O](https://www.tutorialspoint.com/java/java_files_io.htm)
* [Java - Exceptions](https://www.tutorialspoint.com/java/java_exceptions.htm)
* [Java - Inner classes](https://www.tutorialspoint.com/java/java_innerclasses.htm)

Java - Overview

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

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

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

Java is −

* **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
* **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

History of Java

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

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

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

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

Tools You Will Need

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

You will also need the following softwares −

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

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

Java - Environment Setup

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

Try the following example using our online compiler available at [CodingGround](https://www.tutorialspoint.com/codingground.htm)

public class MyFirstJavaProgram {

public static void main(String []args) {

System.out.println("Hello World");

}

}

Local Environment Setup

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

Java SE is freely available from the link [Download Java](https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html" \t "_blank). You can download a version based on your operating system.

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

Setting Up the Path for Windows

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

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

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

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

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

Popular Java Editors

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

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

Java - Basic Syntax

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

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

First Java Program

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

Example

public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\*/

public static void main(String []args) {

System.out.println("Hello World"); // prints Hello World

}

}

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

* Open notepad and add the code as above.
* Save the file as: MyFirstJavaProgram.java.
* Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
* Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
* Now, type ' java MyFirstJavaProgram ' to run your program.
* You will be able to see ' Hello World ' printed on the window.

Output

C:\> javac MyFirstJavaProgram.java

C:\> java MyFirstJavaProgram

Hello World

Basic Syntax

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

* **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
* **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

**Example:** *class MyFirstJavaClass*

* **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

**Example:** *public void myMethodName()*

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

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

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

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

Java Identifiers

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

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

* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character, identifiers can have any combination of characters.
* A key word cannot be used as an identifier.
* Most importantly, identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

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

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

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

Java Variables

Following are the types of variables in Java −

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

Java Arrays

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

Java Enums

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

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

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

Example

class FreshJuice {

enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }

FreshJuiceSize size;

}

public class FreshJuiceTest {

public static void main(String args[]) {

FreshJuice juice = new FreshJuice();

juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;

System.out.println("Size: " + juice.size);

}

}

The above example will produce the following result −

Output

Size: MEDIUM

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

Java Keywords

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

abstract, final,int,byte,float..etc

Inheritance

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

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

Interfaces

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

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

Java - Object and Classes

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

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

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

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

Objects in Java

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

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

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

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

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

Classes in Java

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

Following is a sample of a class.

Example

public class Dog {

String breed;

int ageC

String color;

void barking() {

}

void hungry() {

}

void sleeping() {

}

}

A class can contain any of the following variable types.

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

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

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

Constructors

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

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

Following is an example of a constructor −

Example

public class Puppy {

public Puppy() {

}

public Puppy(String name) {

// This constructor has one parameter, *name*.

}

}

Java also supports [Singleton Classes](https://www.tutorialspoint.com/java/java_using_singleton.htm) where you would be able to create only one instance of a class.

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

Creating an Object

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

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

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

Following is an example of creating an object −

Example

public class Puppy {

public Puppy(String name) {

// This constructor has one parameter, *name*.

System.out.println("Passed Name is :" + name );

}

public static void main(String []args) {

// Following statement would create an object myPuppy

Puppy myPuppy = new Puppy( "tommy" );

}

}

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

Output

Passed Name is :tommy

Accessing Instance Variables and Methods

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

/\* First create an object \*/

ObjectReference = new Constructor();

/\* Now call a variable as follows \*/

ObjectReference.variableName;

/\* Now you can call a class method as follows \*/

ObjectReference.MethodName();

Example

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

public class Puppy {

int puppyAge;

public Puppy(String name) {

// This constructor has one parameter, *name*.

System.out.println("Name chosen is :" + name );

}

public void setAge( int age ) {

puppyAge = age;

}

public int getAge( ) {

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

return puppyAge;

}

public static void main(String []args) {

/\* Object creation \*/

Puppy myPuppy = new Puppy( "tommy" );

/\* Call class method to set puppy's age \*/

myPuppy.setAge( 2 );

/\* Call another class method to get puppy's age \*/

myPuppy.getAge( );

/\* You can access instance variable as follows as well \*/

System.out.println("Variable Value :" + myPuppy.puppyAge );

}

}

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

Output

Name chosen is :tommy

Puppy's age is :2

Variable Value :2

Source File Declaration Rules

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

* There can be only one public class per source file.
* A source file can have multiple non-public classes.
* The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: the class name is *public class Employee{}* then the source file should be as Employee.java.
* If the class is defined inside a package, then the package statement should be the first statement in the source file.
* If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
* Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

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

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

Java Package

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

Import Statements

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

For example, the following line would ask the compiler to load all the classes available in directory java\_installation/java/io −

import java.io.\*;

A Simple Case Study

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

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

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

Example

import java.io.\*;

public class Employee {

String name;

int age;

String designation;

double salary;

// This is the constructor of the class Employee

public Employee(String name) {

this.name = name;

}

// Assign the age of the Employee to the variable age.

public void empAge(int empAge) {

age = empAge;

}

/\* Assign the designation to the variable designation.\*/

public void empDesignation(String empDesig) {

designation = empDesig;

}

/\* Assign the salary to the variable salary.\*/

public void empSalary(double empSalary) {

salary = empSalary;

}

/\* Print the Employee details \*/

public void printEmployee() {

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

System.out.println("Age:" + age );

System.out.println("Designation:" + designation );

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

}

}

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

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

Save the following code in EmployeeTest.java file.

import java.io.\*;

public class EmployeeTest {

public static void main(String args[]) {

/\* Create two objects using constructor \*/

Employee empOne = new Employee("James Smith");

Employee empTwo = new Employee("Mary Anne");

// Invoking methods for each object created

empOne.empAge(26);

empOne.empDesignation("Senior Software Engineer");

empOne.empSalary(1000);

empOne.printEmployee();

empTwo.empAge(21);

empTwo.empDesignation("Software Engineer");

empTwo.empSalary(500);

empTwo.printEmployee();

}

}

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

Output

C:\> javac Employee.java

C:\> javac EmployeeTest.java

C:\> java EmployeeTest

Name:James Smith

Age:26

Designation:Senior Software Engineer

Salary:1000.0

Name:Mary Anne

Age:21

Designation:Software Engineer

Salary:500.0

Java - Basic Datatypes

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

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

There are two data types available in Java −

* Primitive Data Types
* Reference/Object Data Types
* Primitive Data Types:
* byte,short,int,long,float,double,boolean,char

Reference Datatypes

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

Java Literals

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

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

byte a = 68;

char a = 'A'

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

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

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

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

Example

"Hello World"

"two\nlines"

"\"This is in quotes\""

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

char a = '\u0001';

String a = "\u0001";

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

\n,\r,\f,\b

Java - Variable Types

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

You must declare all variables before they can be used. Following is the basic form of a variable declaration −

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

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

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

Example

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

int a = 10, b = 10; // Example of initialization

byte B = 22; // initializes a byte type variable B.

double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a'; // the char variable a iis initialized with value 'a'

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

* Local variables
* Instance variables
* Class/Static variables

Local Variables

* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
* Access modifiers cannot be used for local variables.
* Local variables are visible only within the declared method, constructor, or block.
* Local variables are implemented at stack level internally.
* There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

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

public class Test {

public void pupAge() {

int age = 0;

age = age + 7;

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

}

public static void main(String args[]) {

Test test = new Test();

test.pupAge();

}

}

This will produce the following result −

Output

Puppy age is: 7

Example

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

public class Test {

public void pupAge() {

int age;

age = age + 7;

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

}

public static void main(String args[]) {

Test test = new Test();

test.pupAge();

}

}

This will produce the following error while compiling it −

Output

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

age = age + 7;

^

1 error

Instance Variables

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

Example

import java.io.\*;

public class Employee {

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

public String name;

// salary variable is visible in Employee class only.

private double salary;

// The name variable is assigned in the constructor.

public Employee (String empName) {

name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal) {

salary = empSal;

}

// This method prints the employee details.

public void printEmp() {

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

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

}

public static void main(String args[]) {

Employee empOne = new Employee("Ransika");

empOne.setSalary(1000);

empOne.printEmp();

}

}

This will produce the following result −

Output

name : Ransika

salary :1000.0

Class/Static Variables

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

Example

import java.io.\*;

public class Employee {

// salary variable is a private static variable

private static double salary;

// DEPARTMENT is a constant

public static final String DEPARTMENT = "Development ";

public static void main(String args[]) {

salary = 1000;

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

}

}

This will produce the following result −

Output

Development average salary:1000

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

Java - Modifier Types

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

* [Java Access Modifiers](https://www.tutorialspoint.com/java/java_access_modifiers.htm)
* [Non Access Modifiers](https://www.tutorialspoint.com/java/java_nonaccess_modifiers.htm)

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

Example

*public* class className {

// ...

}

*private* boolean myFlag;

*static final* double weeks = 9.5;

*protected static final* int BOXWIDTH = 42;

*public static* void main(String[] arguments) {

// body of method

}

Access Control Modifiers

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

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

Non-Access Modifiers

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

* The *static* modifier for creating class methods and variables.
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

Java - Basic Operators

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

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

instanceof Operator

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

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

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

**Example**

public class Test {

public static void main(String args[]) {

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This will produce the following result −

**Output**

true

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

**Example**

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]) {

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This will produce the following result −

**Output**

true

Precedence of Java Operators

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

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

Java - Loop Control

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

|  |  |
| --- | --- |
| **Sr.No.** | **Loop & Description** |
| 1 | [**while loop**](https://www.tutorialspoint.com/java/java_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [**for loop**](https://www.tutorialspoint.com/java/java_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [**do...while loop**](https://www.tutorialspoint.com/java/java_do_while_loop.htm)  Like a while statement, except that it tests the condition at the end of the loop body. |

Loop Control Statements

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

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

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [**break statement**](https://www.tutorialspoint.com/java/java_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [**continue statement**](https://www.tutorialspoint.com/java/java_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

Enhanced for loop in Java

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

Syntax

Following is the syntax of enhanced for loop −

for(declaration : expression) {

// Statements

}

* **Declaration** − The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
* **Expression** − This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example

public class Test {

public static void main(String args[]) {

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

for(int x : numbers ) {

System.out.print( x );

System.out.print(",");

}

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

String [] names = {"James", "Larry", "Tom", "Lacy"};

for( String name : names ) {

System.out.print( name );

System.out.print(",");

}

}

}

This will produce the following result −

Output

10, 20, 30, 40, 50,

James, Larry, Tom, Lacy,

Java - Decision Making

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [**if statement**](https://www.tutorialspoint.com/java/if_statement_in_java.htm)  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | [**if...else statement**](https://www.tutorialspoint.com/java/if_else_statement_in_java.htm)  An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| 3 | [**nested if statement**](https://www.tutorialspoint.com/java/nested_if_statements_in_java.htm)  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| 4 | [**switch statement**](https://www.tutorialspoint.com/java/switch_statement_in_java.htm)  A **switch** statement allows a variable to be tested for equality against a list of values. |

The ? : Operator

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

Exp1 ? Exp2 : Exp3;

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

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

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

.

Java - Numbers Class

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

Example

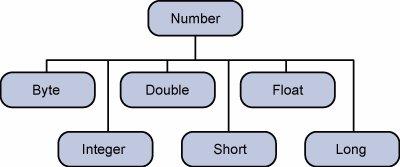
int i = 5000;

float gpa = 13.65;

double mask = 0xaf;

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

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



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

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

Following is an example of boxing and unboxing −

Example

public class Test {

public static void main(String args[]) {

Integer x = 5; // boxes int to an Integer object

x = x + 10; // unboxes the Integer to a int

System.out.println(x);

}

}

This will produce the following result −

Output

15

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

Java - Character Class

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

Example

char ch = 'a';

// Unicode for uppercase Greek omega character

char uniChar = '\u039A';

// an array of chars

char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };

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

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

Character ch = new Character('a');

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

Example

// Here following primitive char 'a'

// is boxed into the Character object ch

Character ch = 'a';

// Here primitive 'x' is boxed for method test,

// return is unboxed to char 'c'

char c = test('x');

Character Methods

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

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | [**isLetter()**](https://www.tutorialspoint.com/java/character_isletter.htm)  Determines whether the specified char value is a letter. |
| 2 | [**isDigit()**](https://www.tutorialspoint.com/java/character_isdigit.htm)  Determines whether the specified char value is a digit. |
| 3 | [**isWhitespace()**](https://www.tutorialspoint.com/java/character_iswhitespace.htm)  Determines whether the specified char value is white space. |
| 4 | [**isUpperCase()**](https://www.tutorialspoint.com/java/character_isuppercase.htm)  Determines whether the specified char value is uppercase. |
| 5 | [**isLowerCase()**](https://www.tutorialspoint.com/java/character_islowercase.htm)  Determines whether the specified char value is lowercase. |
| 6 | [**toUpperCase()**](https://www.tutorialspoint.com/java/character_touppercase.htm)  Returns the uppercase form of the specified char value. |
| 7 | [**toLowerCase()**](https://www.tutorialspoint.com/java/character_tolowercase.htm)  Returns the lowercase form of the specified char value. |
| 8 | [**toString()**](https://www.tutorialspoint.com/java/character_tostring.htm)  Returns a String object representing the specified character value that is, a one-character string. |

For a complete list of methods, please refer to the java.lang.Character API specification.

.

Java - Strings Class

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

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

Creating Strings

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

String greeting = "Hello world!";

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

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

Example

public class StringDemo {

public static void main(String args[]) {

char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };

String helloString = new String(helloArray);

System.out.println( helloString );

}

}

This will produce the following result −

Output

hello.

**Note** − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use [String Buffer & String Builder](https://www.tutorialspoint.com/java/java_string_buffer.htm)Classes.

String Length

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

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

Example

public class StringDemo {

public static void main(String args[]) {

String palindrome = "Dot saw I was Tod";

int len = palindrome.length();

System.out.println( "String Length is : " + len );

}

}

This will produce the following result −

Output

String Length is : 17

Concatenating Strings

The String class includes a method for concatenating two strings −

string1.concat(string2);

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

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

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

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

which results in −

"Hello, world!"

Let us look at the following example −

Example

public class StringDemo {

public static void main(String args[]) {

String string1 = "saw I was ";

System.out.println("Dot " + string1 + "Tod");

}

}

This will produce the following result −

Output

Dot saw I was Tod

Creating Format Strings

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

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

Example

System.out.printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

You can write −

String fs;

fs = String.format("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

System.out.println(fs);

String Methods

Here is the list of methods supported by String class −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |

- Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

dataType[] arrayRefVar; // preferred way.

or

dataType arrayRefVar[]; // works but not preferred way.

**Note** − The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax −

double[] myList; // preferred way.

or

double myList[]; // works but not preferred way.

Creating Arrays

You can create an array by using the new operator with the following syntax −

Syntax

arrayRefVar = new dataType[arraySize];

The above statement does two things −

* It creates an array using new dataType[arraySize].
* It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows −

dataType[] arrayRefVar = {value0, value1, ..., valuek};

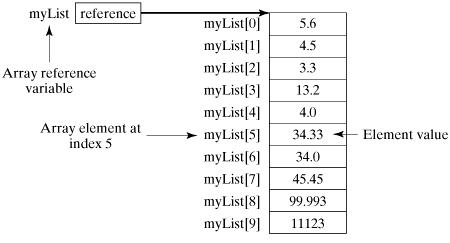
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

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

double[] myList = new double[10];

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



Processing Arrays

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

Example

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

public class TestArray {

public static void main(String[] args) {

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

// Print all the array elements

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

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

}

// Summing all elements

double total = 0;

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

total += myList[i];

}

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

// Finding the largest element

double max = myList[0];

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

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

}

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

}

}

This will produce the following result −

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

Passing Arrays to Methods

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

Example

public static void printArray(int[] array) {

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

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

}

}

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

Example

printArray(new int[]{3, 1, 2, 6, 4, 2});

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array −

Example

public static int[] reverse(int[] list) {

int[] result = new int[list.length];

for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {

result[j] = list[i];

}

return result;

}

Getting Current Date and Time

This is a very easy method to get current date and time in Java. You can use a simple Date object with *toString()* method to print the current date and time as follows −

Example

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display time and date using toString()

System.out.println(date.toString());

}

}

This will produce the following result −

Output

on May 04 09:51:52 CDT 2009

Parsing Strings into Dates

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

Example

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

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

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

Date t;

try {

t = ft.parse(input);

System.out.println(t);

}catch (ParseException e) {

System.out.println("Unparseable using " + ft);

}

}

}

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

Output

1818-11-11 Parses as Wed Nov 11 00:00:00 EST 1818

Sleeping for a While

You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, the following program would sleep for 3 seconds −

Example

import java.util.\*;

public class SleepDemo {

public static void main(String args[]) {

try {

System.out.println(new Date( ) + "\n");

Thread.sleep(5\*60\*10);

System.out.println(new Date( ) + "\n");

}catch (Exception e) {

System.out.println("Got an exception!");

}

}

}

This will produce the following result −

Output

Sun May 03 18:04:41 GMT 2009

Sun May 03 18:04:51 GMT 2009

Measuring Elapsed Time

Sometimes, you may need to measure point in time in milliseconds. So let's re-write the above example once again −

Example

import java.util.\*;

public class DiffDemo {

public static void main(String args[]) {

try {

long start = System.currentTimeMillis( );

System.out.println(new Date( ) + "\n");

Thread.sleep(5\*60\*10);

System.out.println(new Date( ) + "\n");

long end = System.currentTimeMillis( );

long diff = end - start;

System.out.println("Difference is : " + diff);

}catch (Exception e) {

System.out.println("Got an exception!");

}

}

}

This will produce the following result −

Output

Sun May 03 18:16:51 GMT 2009

Sun May 03 18:16:57 GMT 2009

Difference is : 5993

GregorianCalendar Class

GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. We did not discuss Calendar class in this tutorial, you can look up standard Java documentation for this.

The **getInstance( )** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

Example

import java.util.\*;

public class GregorianCalendarDemo {

public static void main(String args[]) {

String months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",

"Oct", "Nov", "Dec"};

int year;

// Create a Gregorian calendar initialized

// with the current date and time in the

// default locale and timezone.

GregorianCalendar gcalendar = new GregorianCalendar();

// Display current time and date information.

System.out.print("Date: ");

System.out.print(months[gcalendar.get(Calendar.MONTH)]);

System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");

System.out.println(year = gcalendar.get(Calendar.YEAR));

System.out.print("Time: ");

System.out.print(gcalendar.get(Calendar.HOUR) + ":");

System.out.print(gcalendar.get(Calendar.MINUTE) + ":");

System.out.println(gcalendar.get(Calendar.SECOND));

// Test if the current year is a leap year

if(gcalendar.isLeapYear(year)) {

System.out.println("The current year is a leap year");

}else {

System.out.println("The current year is not a leap year");

}

}

}

This will produce the following result −

Output

Date: Apr 22 2009

Time: 11:25:27

The current year is not a leap year

For a complete list of constant available in Calendar class, you can refer the standard Java documentation.

Java - Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, for example, the system actually executes several statements in order to display a message on the console.

Creating Method

Considering the following example to explain the syntax of a method −

**Syntax**

public static int methodName(int a, int b) {

// body

}

Here,

* **public static** − modifier
* **int** − return type
* **methodName** − name of the method
* **a, b** − formal parameters
* **int a, int b** − list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax −

**Syntax**

modifier returnType nameOfMethod (Parameter List) {

// method body

}

The syntax shown above includes −

* **modifier** − It defines the access type of the method and it is optional to use.
* **returnType** − Method may return a value.
* **nameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.
* **Parameter List** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
* **method body** − The method body defines what the method does with the statements.

**Example**

Here is the source code of the above defined method called **max()**. This method takes two parameters num1 and num2 and returns the maximum between the two −

/\*\* the snippet returns the minimum between two numbers \*/

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when −

* the return statement is executed.
* it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example −

System.out.println("This is tutorialspoint.com!");

The method returning value can be understood by the following example −

int result = sum(6, 9);

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

**Example**

public class ExampleMinNumber {

public static void main(String[] args) {

int a = 11;

int b = 6;

int c = minFunction(a, b);

System.out.println("Minimum Value = " + c);

}

/\*\* returns the minimum of two numbers \*/

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

}

This will produce the following result −

**Output**

Minimum value = 6

The void Keyword

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown in the following example.

**Example**

public class ExampleVoid {

public static void main(String[] args) {

methodRankPoints(255.7);

}

public static void methodRankPoints(double points) {

if (points >= 202.5) {

System.out.println("Rank:A1");

}else if (points >= 122.4) {

System.out.println("Rank:A2");

}else {

System.out.println("Rank:A3");

}

}

}

This will produce the following result −

**Output**

Rank:A1

Passing Parameters by Value

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

**Example**

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

public class swappingExample {

public static void main(String[] args) {

int a = 30;

int b = 45;

System.out.println("Before swapping, a = " + a + " and b = " + b);

// Invoke the swap method

swapFunction(a, b);

System.out.println("\n\*\*Now, Before and After swapping values will be same here\*\*:");

System.out.println("After swapping, a = " + a + " and b is " + b);

}

public static void swapFunction(int a, int b) {

System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

// Swap n1 with n2

int c = a;

a = b;

b = c;

System.out.println("After swapping(Inside), a = " + a + " b = " + b);

}

}

This will produce the following result −

**Output**

Before swapping, a = 30 and b = 45

Before swapping(Inside), a = 30 b = 45

After swapping(Inside), a = 45 b = 30

\*\*Now, Before and After swapping values will be same here\*\*:

After swapping, a = 30 and b is 45

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let’s consider the example discussed earlier for finding minimum numbers of integer type. If, let’s say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same −

**Example**

public class ExampleOverloading {

public static void main(String[] args) {

int a = 11;

int b = 6;

double c = 7.3;

double d = 9.4;

int result1 = minFunction(a, b);

// same function name with different parameters

double result2 = minFunction(c, d);

System.out.println("Minimum Value = " + result1);

System.out.println("Minimum Value = " + result2);

}

// for integer

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

// for double

public static double minFunction(double n1, double n2) {

double min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

}

This will produce the following result −

**Output**

Minimum Value = 6

Minimum Value = 7.3

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

Using Command-Line Arguments

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

**Example**

The following program displays all of the command-line arguments that it is called with −

public class CommandLine {

public static void main(String args[]) {

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

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

}

}

}

Try executing this program as shown here −

$java CommandLine this is a command line 200 -100

This will produce the following result −

**Output**

args[0]: this

args[1]: is

args[2]: a

args[3]: command

args[4]: line

args[5]: 200

args[6]: -100

The Constructors

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

**Example**

Here is a simple example that uses a constructor without parameters −

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass() {

x = 10;

}

}

You will have to call constructor to initialize objects as follows −

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

System.out.println(t1.x + " " + t2.x);

}

}

**Output**

10 10

Parameterized Constructor

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

**Example**

Here is a simple example that uses a constructor with a parameter −

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass(int i ) {

x = i;

}

}

You will need to call a constructor to initialize objects as follows −

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass( 10 );

MyClass t2 = new MyClass( 20 );

System.out.println(t1.x + " " + t2.x);

}

}

This will produce the following result −

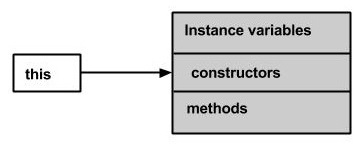
**Output**

10 20

The this keyword

**this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

**Note** − The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to −

* Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

class Student {

int age;

Student(int age) {

this.age = age;

}

}

* Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

class Student {

int age

Student() {

this(20);

}

Student(int age) {

this.age = age;

}

}

**Example**

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the following program in a file with the name, **This\_Example.java**.

public class This\_Example {

// Instance variable num

int num = 10;

This\_Example() {

System.out.println("This is an example program on keyword this");

}

This\_Example(int num) {

// Invoking the default constructor

this();

// Assigning the local variable *num* to the instance variable *num*

this.num = num;

}

public void greet() {

System.out.println("Hi Welcome to Tutorialspoint");

}

public void print() {

// Local variable num

int num = 20;

// Printing the instance variable

System.out.println("value of local variable num is : "+num);

// Printing the local variable

System.out.println("value of instance variable num is : "+this.num);

// Invoking the greet method of a class

this.greet();

}

public static void main(String[] args) {

// Instantiating the class

This\_Example obj1 = new This\_Example();

// Invoking the print method

obj1.print();

// Passing a new value to the num variable through parametrized constructor

This\_Example obj2 = new This\_Example(30);

// Invoking the print method again

obj2.print();

}

}

This will produce the following result −

**Output**

This is an example program on keyword this

value of local variable num is : 20

value of instance variable num is : 10

Hi Welcome to Tutorialspoint

This is an example program on keyword this

value of local variable num is : 20

value of instance variable num is : 30

Hi Welcome to Tutorialspoint

Variable Arguments(var-args)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows −

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

**Example**

public class VarargsDemo {

public static void main(String args[]) {

// Call method with variable args

printMax(34, 3, 3, 2, 56.5);

printMax(new double[]{1, 2, 3});

}

public static void printMax( double... numbers) {

if (numbers.length == 0) {

System.out.println("No argument passed");

return;

}

double result = numbers[0];

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

if (numbers[i] > result)

result = numbers[i];

System.out.println("The max value is " + result);

}

}

This will produce the following result −

**Output**

The max value is 56.5

The max value is 3.0

The finalize( ) Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

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

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.

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

The finalize( ) method has this general form −

protected void finalize( ) {

// finalization code here

}

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

This means that you cannot know when or even if finalize( ) will be executed. For example, if your program ends before garbage collection occurs, finalize( ) will not execute.

Java - Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

* **InPutStream** − The InputStream is used to read data from a source.
* **OutPutStream** − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Java - Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

* **Checked exceptions** − A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

If you try to compile the above program, you will get the following exceptions.

Output

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

**Note** − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

* **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

public class Unchecked\_Demo {

public static void main(String args[]) {

int num[] = {1, 2, 3, 4};

System.out.println(num[5]);

}

}

If you compile and execute the above program, you will get the following exception.

Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

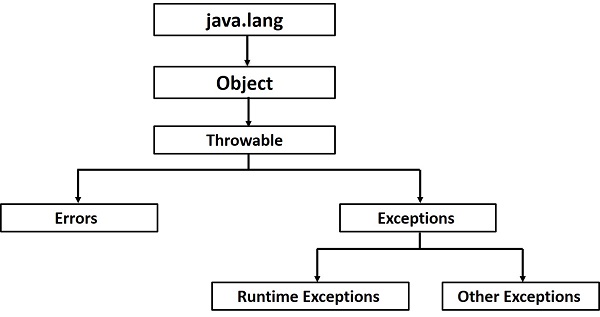
* **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Following is a list of most common checked and unchecked [Java's Built-in Exceptions](https://www.tutorialspoint.com/java/java_builtin_exceptions.htm).

Exceptions Methods

Following is the list of important methods available in the Throwable class.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public String getMessage()**  Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | **public Throwable getCause()**  Returns the cause of the exception as represented by a Throwable object. |
| 3 | **public String toString()**  Returns the name of the class concatenated with the result of getMessage(). |
| 4 | **public void printStackTrace()**  Prints the result of toString() along with the stack trace to System.err, the error output stream. |
|  |  |
|  |  |

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 follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

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

// File Name : ExcepTest.java

import java.io.\*;

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

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

}catch(ArrayIndexOutOfBoundsException e) {

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

}

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

}

}

This will produce the following result −

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;

}

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it −

catch (IOException|FileNotFoundException ex) {

logger.log(ex);

throw ex;

The Throws/Throw Keywords

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

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

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

The following method declares that it throws a RemoteException −

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.

The try-with-resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

public class ReadData\_Demo {

public static void main(String args[]) {

FileReader fr = null;

try {

File file = new File("file.txt");

fr = new FileReader(file); char [] a = new char[50];

fr.read(a); // reads the content to the array

for(char c : a)

System.out.print(c); // prints the characters one by one

}catch(IOException e) {

e.printStackTrace();

}finally {

try {

fr.close();

}catch(IOException ex) {

ex.printStackTrace();

}

}

}

}

**try-with-resources**, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

try(FileReader fr = new FileReader("file path")) {

// use the resource

}catch() {

// body of catch

}

}

Following is the program that reads the data in a file using try-with-resources statement.

Example

import java.io.FileReader;

import java.io.IOException;

public class Try\_withDemo {

public static void main(String args[]) {

try(FileReader fr = new FileReader("E://file.txt")) {

char [] a = new char[50];

fr.read(a); // reads the contentto the array

for(char c : a)

System.out.print(c); // prints the characters one by one

}catch(IOException e) {

e.printStackTrace();

}

}

}

Following points are to be kept in mind while working with try-with-resources statement.

* To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
* You can declare more than one class in try-with-resources statement.
* While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
* Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
* The resource declared in try gets instantiated just before the start of the try-block.
* The resource declared at the try block is implicitly declared as final.

User-defined Exceptions

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

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

We can define our own Exception class as below −

class MyException extends Exception {

}

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

Example

// File Name InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception {

private double amount;

public InsufficientFundsException(double amount) {

this.amount = amount;

}

public double getAmount() {

return amount;

}

}

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

// File Name CheckingAccount.java

import java.io.\*;

public class CheckingAccount {

private double balance;

private int number;

public CheckingAccount(int number) {

this.number = number;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) throws InsufficientFundsException {

if(amount <= balance) {

balance -= amount;

}else {

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

public double getBalance() {

return balance;

}

public int getNumber() {

return number;

}

}

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

// File Name BankDemo.java

public class BankDemo {

public static void main(String [] args) {

CheckingAccount c = new CheckingAccount(101);

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

c.deposit(500.00);

try {

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

c.withdraw(100.00);

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

c.withdraw(600.00);

}catch(InsufficientFundsException e) {

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

e.printStackTrace();

}

}

}

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

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)

Common Exceptions

In Java, it is possible to define two catergories of Exceptions and Errors.

* **JVM Exceptions** − These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
* **Programmatic Exceptions** − These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

Java - Inner classes

In this chapter, we will discuss inner classes of Java.

Nested Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

**Syntax**

Following is the syntax to write a nested class. Here, the class **Outer\_Demo** is the outer class and the class **Inner\_Demo** is the nested class.

class Outer\_Demo {

class Nested\_Demo {

}

}

Nested classes are divided into two types −

* **Non-static nested classes** − These are the non-static members of a class.
* **Static nested classes** − These are the static members of a class.



Inner Classes (Non-static Nested Classes)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are −

* Inner Class
* Method-local Inner Class
* Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

**Example**

class Outer\_Demo {

int num;

// inner class

private class Inner\_Demo {

public void print() {

System.out.println("This is an inner class");

}

}

// Accessing he inner class from the method within

void display\_Inner() {

Inner\_Demo inner = new Inner\_Demo();

inner.print();

}

}

public class My\_class {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer = new Outer\_Demo();

// Accessing the display\_Inner() method.

outer.display\_Inner();

}

}

Here you can observe that **Outer\_Demo** is the outer class, **Inner\_Demo** is the inner class, **display\_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

If you compile and execute the above program, you will get the following result −

**Output**

This is an inner class.

Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

Outer\_Demo outer = new Outer\_Demo();

Outer\_Demo.Inner\_Demo inner = outer.new Inner\_Demo();

The following program shows how to access the private members of a class using inner class.

**Example**

class Outer\_Demo {

// private variable of the outer class

private int num = 175;

// inner class

public class Inner\_Demo {

public int getNum() {

System.out.println("This is the getnum method of the inner class");

return num;

}

}

}

public class My\_class2 {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer = new Outer\_Demo();

// Instantiating the inner class

Outer\_Demo.Inner\_Demo inner = outer.new Inner\_Demo();

System.out.println(inner.getNum());

}

}

If you compile and execute the above program, you will get the following result −

**Output**

The value of num in the class Test is: 175

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

**Example**

public class Outerclass {

// instance method of the outer class

void my\_Method() {

int num = 23;

// method-local inner class

class MethodInner\_Demo {

public void print() {

System.out.println("This is method inner class "+num);

}

} // end of inner class

// Accessing the inner class

MethodInner\_Demo inner = new MethodInner\_Demo();

inner.print();

}

public static void main(String args[]) {

Outerclass outer = new Outerclass();

outer.my\_Method();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is method inner class 23

Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows −

**Syntax**

AnonymousInner an\_inner = new AnonymousInner() {

public void my\_method() {

........

........

}

};

The following program shows how to override the method of a class using anonymous inner class.

**Example**

abstract class AnonymousInner {

public abstract void mymethod();

}

public class Outer\_class {

public static void main(String args[]) {

AnonymousInner inner = new AnonymousInner() {

public void mymethod() {

System.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is an example of anonymous inner class

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument −

obj.my\_Method(new My\_Class() {

public void Do() {

.....

.....

}

});

The following program shows how to pass an anonymous inner class as a method argument.

**Example**

// interface

interface Message {

String greet();

}

public class My\_class {

// method which accepts the object of interface Message

public void displayMessage(Message m) {

System.out.println(m.greet() +

", This is an example of anonymous inner class as an argument");

}

public static void main(String args[]) {

// Instantiating the class

My\_class obj = new My\_class();

// Passing an anonymous inner class as an argument

obj.displayMessage(new Message() {

public String greet() {

return "Hello";

}

});

}

}

If you compile and execute the above program, it gives you the following result −

**Output**

Hello This is an example of anonymous inner class as an argument

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows −

**Syntax**

class MyOuter {

static class Nested\_Demo {

}

}

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

**Example**

public class Outer {

static class Nested\_Demo {

public void my\_method() {

System.out.println("This is my nested class");

}

}

public static void main(String args[]) {

Outer.Nested\_Demo nested = new Outer.Nested\_Demo();

nested.my\_method();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is my nested class