Java Basic Tutorial

28 January 2022 12:30

© RAJAT KUMAR
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Following : https://www.w3schools.com/java/default.asp

Java Tutorial

Printing Data :-

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

Note: Java is case-sensitive: "MyClass" and "myclass" has different meaning.

Primitive Data Types
A primitive data type specifies the size and type of variable values, and it has no additional methods.

Non-Primitive Data Types
Non-primitive data types are called reference types because they refer to objects.

Widening Casting (automatically) - converting a smaller type to a larger type size
Widening casting is done automatically when passing a smaller size type to a larger size type:
byte -> short -> char -> int -> long -> float -> double

Narrowing Casting (manually) - converting a larger type to a smaller size type
Narrowing casting must be done manually by placing the type in parentheses in front of the value:
double -> float -> long -> int -> char -> short -> byte

Java divides the operators into the following groups:
Arithmetic operators
Assignment operators
Comparison operators
Logical operators
Bitwise operators

Java Strings
Use length() method to find length;

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println(txt.length());

System.out.println(txt.toUpperCase());
```

Txt.indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace):

concat() method to concatenate two strings:

Java Math
Math.max(x,y)
Math.min(x,y)
Math.sqrt(x)
Math.abs(x)
Math.random()

Short Hand If...Else (Ternary Operator)

```
variable = (condition) ? expressionTrue :  expressionFalse;
String result = (time < 18) ? "Good day." : "Good evening.";
```

Java Switch

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}

int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

- If a Class is public, it should be in a file whose name is same as class name
- Static function don't require to create an object to call them.
- If we don't want to create object from a class, we make it abstract.
  Public abstract class Account {

  }
- Now we will not be able to create object from the account class.
- But we can create objects for classes that inherit account class

- Interface :
  public interface IBaseRate {

  }
- Child class :
  public class Savings extends Account {

  }
- Parent class : (we can't create object for this)
  public abstract class Account implements IBaseRate {

  }
- Abstract methods must be implement in derive classes.

The Do/While Loop

```
do {
  // code block to be executed
}
while (condition);
```

```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

For-Each Loop

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

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

Java Arrays

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

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

```
Loop Through an Array with For-Each
for (type variable : arrayname) {
  ...
}
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

```
int[] myNum = {10, 20, 30, 40};
```

Multidimensional Arrays

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

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

Java Methods

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

```
// Outputs "I just got executed!"
```

Example Explained
myMethod() is the name of the method
static means that the method belongs to the Main class and not an object of the Main class.
void means that this method does not have a return value.

Java Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

Java OOP

OOP stands for Object-Oriented Programming

Classes and objects are the two main aspects of object-oriented programming.
Class - Fruit
Objects - Apple, Banana, Mango

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

Java Classes and Objects

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

If you don't want the ability to override existing values, declare the attribute as final:

Java Class Methods

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

Static vs. Non-Static
We will often see Java programs that have either static or public attributes and methods.

In the example above, we created a static method,
which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

The dot (.) is used to access the object's attributes and methods.

Java Constructors
A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

```
// Create a Main class
public class Main {
  int x;  // Create a class attribute

  // Create a class constructor for the Main class
  public Main() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    Main myObj = new Main(); // Create an object of class Main (This will call the constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}

// Outputs 5
```

Constructor Parameters
Constructors can also take parameters, which is used to initialize attributes.

```
public class Main {
  int modelYear;
  String modelName;

  public Main(int year, String name) {
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}

// Outputs 1969 Mustang
```

Java Modifiers // https://www.w3schools.com/java/java_modifiers.asp
We divide modifiers into two groups:

Access Modifiers - controls the access level
Non-Access Modifiers - do not control access level, but provides other functionality

Access Modifiers
For classes, we can use either public or default:
Public : The class is accessible by any other class

default : The class is only accessible by classes in the same package. This is used when we don't specify a modifier.

For attributes, methods and constructors, you can use the one of the following:
Public : The code is accessible for all classes
Private : The code is only accessible within the declared class
Default : The code is only accessible in the same package. This is used when you don't specify a modifier.
Protected : The code is accessible in the same package and subclasses.

Non-Access Modifiers
For classes, we can use either final or abstract:
Final : The class cannot be inherited by other classes
Abstract : The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.

For attributes and methods, we can use the one of the following:
Final : Attributes and methods cannot be overridden/modified
Static : Attributes and methods belongs to the class, rather than an object
Abstract : Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example abstract void run();. The body is provided by the subclass (inherited from).
Transient : Attributes and methods are skipped when serializing the object containing them
Synchronized : Methods can only be accessed by one thread at a time
Volatile : The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Final
If you don't want the ability to override existing attribute values, declare attributes as final:

Static
A static method means that it can be accessed without creating an object of the class, unlike public:

Abstract
An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass:

```
// Code from filename: Main.java
// abstract class
abstract class Main {
  public String fname = "John";
  public int age = 24;
  public abstract void study(); // abstract method
}

// Subclass (inherit from Main)
class Student extends Main {
  public int graduationYear = 2018;
  public void study() { // the body of the abstract method is provided here
    System.out.println("Studying all day long");
  }
}
// End code from filename: Main.java

// Code from filename: Second.java
class Second {
  public static void main(String[] args) {
    // create an object of the Student class (which inherits attributes and methods from Main)
    Student myObj = new Student();

    System.out.println("Name: " + myObj.fname);
    System.out.println("Age: " + myObj.age);
    System.out.println("Graduation Year: " + myObj.graduationYear);
    myObj.study(); // call abstract method
  }
}
```

Encapsulation
The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, we must:

- declare class variables/attributes as private
- provide public get and set methods to access and update the value of a private variable

Get and Set :-
The get method returns the variable value, and the set method sets the value.

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

The get method returns the value of the variable name.
The set method takes a parameter (newName) and assigns it to the name variable. The this keyword is used to refer to the current object.

Why Encapsulation?

- Better control of class attributes and methods

- Class attributes can be made read-only (if you only use the get method), or write-only (if you only use the set method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Java Packages & API
A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages
```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class
```
import java.util.Scanner;    // Taking Input from user

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

User-defined Packages
```
// MyPackageClass.java

package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Java Inheritance (Subclass and Superclass)
it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- subclass (child) - the class that inherits from another class
- superclass (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

```
class Vehicle {
  protected String brand = "Ford";       // Vehicle attribute
  public void honk() {                   // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

The final Keyword
If you don't want other classes to inherit from a class, use the final keyword:

Java Polymorphism
Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance

Inheritance lets us inherit attributes and methods from another class.
Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
```

```
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

Why And When To Use "Inheritance" and "Polymorphism"?
- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Java Inner Classes
it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
To access the inner class, create an object of the outer class, and then create an object of the inner class:

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

Private Inner Class
Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

Static Inner Class
An inner class can also be static, which means that you can access it without creating an object of the outer class:

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

Java Abstraction
Data abstraction is the process of hiding certain details and showing only essential information to the user.

The *abstract* keyword is a non-access modifier, used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the

Polymorphism to an abstract class:

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Interfaces
An interface is a completely "abstract class" that is used to group related methods with empty bodies:

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class:

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Like abstract classes, interfaces cannot be used to create objects

Multiple Interfaces

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Enums
An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}

Level myVar = Level.MEDIUM;

for (Level myVar : Level.values()) {
  System.out.println(myVar);
}
```

Java User Input (Scanner)

```
import java.util.Scanner;  // Import the Scanner class

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

Java Date and Time

```
import java.time.LocalDate; // import the LocalDate class

public class Main {
  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now(); // Create a date object
    System.out.println(myObj); // Display the current date
  }
}
```

LocalDate	Represents a date (year, month, day (yyyy-MM-dd))
LocalTime	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
LocalDateTime	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
DateTimeFormatter	Formatter for displaying and parsing date-time objects

Java ArrayList

The ArrayList class is a resizable array.
The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want.

```
ArrayList<String> cars = new ArrayList<>(); // Create an ArrayList object

cars.add("Mazda");
cars.get(0);
cars.set(0, "Opel");
cars.remove(0);
cars.clear();
cars.size();

 Collections.sort(cars);  // Sort cars

for (int i = 0; i < cars.size(); i++) {
    System.out.println(cars.get(i));
  }

for (String i : cars) {
    System.out.println(i);
  }

ArrayList<Integer> myNumbers = new ArrayList<Integer>();
for (int i : myNumbers) {
    System.out.println(i);
  }
```

Java LinkedList

The LinkedList class is almost identical to the ArrayList:

```
LinkedList<String> cars = new LinkedList<String>();

cars.add("Volvo");
```

the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently:

addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list

removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

ArrayList vs. LinkedList

The *ArrayList* class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

The *LinkedList* stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

When To Use
Use an ArrayList for storing and accessing data, and LinkedList to manipulate data.

Java HashMap
A HashMap however, store items in "key/value" pairs, and you can access them by an index of another type.
One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

```
HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");
capitalCities.get("England");
capitalCities.remove("England");
capitalCities.clear();
capitalCities.size();

// Print keys
for (String i : capitalCities.keySet()) {
  System.out.println(i);
}

// Print values
for (String i : capitalCities.values()) {
  System.out.println(i);
}

// Print keys and values
for (String i : capitalCities.keySet()) {
  System.out.println("key: " + i + " value: " + capitalCities.get(i));
}

HashMap<String, Integer> people = new HashMap<String, Integer>();
```

Java HashSet
A HashSet is a collection of items where every item is unique

```
HashSet<String> cars = new HashSet<String>();
cars.add("Volvo");
cars.contains("Mazda");
cars.remove("Volvo");
cars.clear();
cars.size();

for (String i : cars) {
  System.out.println(i);
}

HashSet<Integer> numbers = new HashSet<Integer>();
```

Java Iterator
An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

```
ArrayList<String> cars = new ArrayList<String>();
   cars.add("Volvo");
   cars.add("BMW");
   cars.add("Ford");
   cars.add("Mazda");

   // Get the iterator
   Iterator<String> it = cars.iterator();

   // Print the first item
   System.out.println(it.next());

while(it.hasNext()) {
  System.out.println(it.next());
}

Iterator<Integer> it = numbers.iterator();
   while(it.hasNext()) {
     Integer i = it.next();
     if(i < 10) {
       it.remove();
```

```
    }
  }
  System.out.println(numbers);
```

Trying to remove items using a for loop or a for-each loop would not work correctly because the collection is changing size at the same time that the code is trying to loop.

Java Wrapper Classes
Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

Primitive Data Type	:	Wrapper Class
byte		Byte
short		Short
int		Integer
long		Long
float		Float
double		Double
boolean		Boolean
char		Character

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid
ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid


Integer myInt = 5;
  Double myDouble = 5.99;
  Character myChar = 'A';
  System.out.println(myInt);
  System.out.println(myDouble);
  System.out.println(myChar);

  Integer myInt = 5;
  Double myDouble = 5.99;
  Character myChar = 'A';

  System.out.println(myInt);
  System.out.println(myDouble);
  System.out.println(myChar);

  System.out.println(myInt.intValue());
  System.out.println(myDouble.doubleValue());
  System.out.println(myChar.charValue());

Integer myInt = 100;
String myString = myInt.toString();
System.out.println(myString.length());
```

Java Exceptions - Try...Catch
When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error).

Java try and catch
The try statement allows you to define a block of code to be tested for errors while it is being executed.
The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
The try and catch keywords come in pairs:

```
try {
 //  Block of code to try
}
catch(Exception e) {
 //  Block of code to handle errors
}

public class Main {
 public static void main(String[ ] args) {
  try {
   int[] myNumbers = {1, 2, 3};
   System.out.println(myNumbers[10]);
  } catch (Exception e) {
   System.out.println("Something went wrong.");
  }
 }
}
```

Finally
The finally statement lets you execute code, after try...catch, regardless of the result:

```
public class Main {
 public static void main(String[] args) {
  try {
   int[] myNumbers = {1, 2, 3};
   System.out.println(myNumbers[10]);
  } catch (Exception e) {
   System.out.println("Something went wrong.");
  } finally {
   System.out.println("The 'try catch' is finished.");
  }
```

```
  }
}
```

The throw keyword
The throw statement allows you to create a custom error.

The throw statement is used together with an exception type. There are many exception types available in Java:
ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

// Throw an exception if age is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```