# Java

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language. Read more here.

```java
// Single-line comments start with //
/*
Multi-line comments look like this.
*/
/**
JavaDoc comments look like this. Used to describe the Class or various
attributes of a Class.
*/

// Import ArrayList class inside of the java.util package
import java.util.ArrayList;
// Import all classes inside of java.security package
import java.security.*;

// Each .java file contains one outer-level public class, with the same name as
// the file.
public class LearnJava {

    // In order to run a java program, it must have a main method as an entry point.
    public static void main (String[] args) {

        // Use System.out.println() to print lines.
        System.out.println("Hello World!");
        System.out.println(
            "Integer: " + 10 +
            " Double: " + 3.14 +
            " Boolean: " + true);

        // To print without a newline, use System.out.print().
        System.out.print("Hello ");
        System.out.print("World");

        // Use System.out.printf() for easy formatted printing.
        System.out.printf("pi = %.5f", Math.PI); // => pi = 3.14159

        ///////////////////////////////////////
        // Variables
        ///////////////////////////////////////

        /*
        *  Variable Declaration
        */
        // Declare a variable using <type> <name>
        int fooInt;
        // Declare multiple variables of the same type <type> <name1>, <name2>, <name3>
        int fooInt1, fooInt2, fooInt3;

        /*
        *  Variable Initialization
```

```
*/

// Initialize a variable using <type> <name> = <val>
int fooInt = 1;
// Initialize multiple variables of same type with same value <type> <name1>, <name2>, <name3>
int fooInt1, fooInt2, fooInt3;
fooInt1 = fooInt2 = fooInt3 = 1;

/*
*  Variable types
*/
// Byte - 8-bit signed two's complement integer
// (-128 <= byte <= 127)
byte fooByte = 100;

// Short - 16-bit signed two's complement integer
// (-32,768 <= short <= 32,767)
short fooShort = 10000;

// Integer - 32-bit signed two's complement integer
// (-2,147,483,648 <= int <= 2,147,483,647)
int fooInt = 1;

// Long - 64-bit signed two's complement integer
// (-9,223,372,036,854,775,808 <= long <= 9,223,372,036,854,775,807)
long fooLong = 100000L;
// L is used to denote that this variable value is of type Long;
// anything without is treated as integer by default.

// Note: Java has no unsigned types.

// Float - Single-precision 32-bit IEEE 754 Floating Point
// 2^-149 <= float <= (2-2^-23) * 2^127
float fooFloat = 234.5f;
// f or F is used to denote that this variable value is of type float;
// otherwise it is treated as double.

// Double - Double-precision 64-bit IEEE 754 Floating Point
// 2^-1074 <= x <= (2-2^-52) * 2^1023
double fooDouble = 123.4;

// Boolean - true & false
boolean fooBoolean = true;
boolean barBoolean = false;

// Char - A single 16-bit Unicode character
char fooChar = 'A';

// final variables can't be reassigned to another object,
final int HOURS_I_WORK_PER_WEEK = 9001;
// but they can be initialized later.
final double E;
E = 2.71828;
```

```java
// BigInteger - Immutable arbitrary-precision integers
//
// BigInteger is a data type that allows programmers to manipulate
// integers longer than 64-bits. Integers are stored as an array of
// of bytes and are manipulated using functions built into BigInteger
//
// BigInteger can be initialized using an array of bytes or a string.

BigInteger fooBigInteger = new BigInteger(fooByteArray);


// BigDecimal - Immutable, arbitrary-precision signed decimal number
//
// A BigDecimal takes two parts: an arbitrary precision integer
// unscaled value and a 32-bit integer scale
//
// BigDecimal allows the programmer complete control over decimal
// rounding. It is recommended to use BigDecimal with currency values
// and where exact decimal precision is required.
//
// BigDecimal can be initialized with an int, long, double or String
// or by initializing the unscaled value (BigInteger) and scale (int).

BigDecimal fooBigDecimal = new BigDecimal(fooBigInteger, fooInt);

// Be wary of the constructor that takes a float or double as
// the inaccuracy of the float/double will be copied in BigDecimal.
// Prefer the String constructor when you need an exact value.

BigDecimal tenCents = new BigDecimal("0.1");


// Strings
String fooString = "My String Is Here!";

// \n is an escaped character that starts a new line
String barString = "Printing on a new line?\nNo Problem!";
// \t is an escaped character that adds a tab character
String bazString = "Do you want to add a tab?\tNo Problem!";
System.out.println(fooString);
System.out.println(barString);
System.out.println(bazString);

// Arrays
// The array size must be decided upon instantiation
// The following formats work for declaring an array
// <datatype>[] <var name> = new <datatype>[<array size>];
// <datatype> <var name>[] = new <datatype>[<array size>];
int[] intArray = new int[10];
String[] stringArray = new String[1];
boolean boolArray[] = new boolean[100];

// Another way to declare & initialize an array
```

```
int[] y = {9000, 1000, 1337};
String names[] = {"Bob", "John", "Fred", "Juan Pedro"};
boolean bools[] = new boolean[] {true, false, false};

// Indexing an array - Accessing an element
System.out.println("intArray @ 0: " + intArray[0]);

// Arrays are zero-indexed and mutable.
intArray[1] = 1;
System.out.println("intArray @ 1: " + intArray[1]); // => 1

// Others to check out
// ArrayLists - Like arrays except more functionality is offered, and
//              the size is mutable.
// LinkedLists - Implementation of doubly-linked list. All of the
//               operations perform as could be expected for a
//               doubly-linked list.
// Maps - A set of objects that map keys to values. Map is
//        an interface and therefore cannot be instantiated.
//        The type of keys and values contained in a Map must
//        be specified upon instantiation of the implementing
//        class. Each key may map to only one corresponding value,
//        and each key may appear only once (no duplicates).
// HashMaps - This class uses a hashtable to implement the Map
//            interface. This allows the execution time of basic
//            operations, such as get and insert element, to remain
//            constant even for large sets.

///////////////////////////////////////
// Operators
///////////////////////////////////////
System.out.println("\n->Operators");

int i1 = 1, i2 = 2; // Shorthand for multiple declarations

// Arithmetic is straightforward
System.out.println("1+2 = " + (i1 + i2)); // => 3
System.out.println("2-1 = " + (i2 - i1)); // => 1
System.out.println("2*1 = " + (i2 * i1)); // => 2
System.out.println("1/2 = " + (i1 / i2)); // => 0 (int/int returns an int)
System.out.println("1/2 = " + (i1 / (double)i2)); // => 0.5

// Modulo
System.out.println("11%3 = "+(11 % 3)); // => 2

// Comparison operators
System.out.println("3 == 2? " + (3 == 2)); // => false
System.out.println("3 != 2? " + (3 != 2)); // => true
System.out.println("3 > 2? " + (3 > 2)); // => true
System.out.println("3 < 2? " + (3 < 2)); // => false
System.out.println("2 <= 2? " + (2 <= 2)); // => true
System.out.println("2 >= 2? " + (2 >= 2)); // => true

// Boolean operators
```

```
System.out.println("3 > 2 && 2 > 3? " + ((3 > 2) && (2 > 3))); // => false
System.out.println("3 > 2 || 2 > 3? " + ((3 > 2) || (2 > 3))); // => true
System.out.println("!(3 == 2)? " + (!(3 == 2))); // => true

// Bitwise operators!
/*
~       Unary bitwise complement
<<      Signed left shift
>>      Signed/Arithmetic right shift
>>>     Unsigned/Logical right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
*/

// Incrementations
int i = 0;
System.out.println("\n->Inc/Dec-rementation");
// The ++ and -- operators increment and decrement by 1 respectively.
// If they are placed before the variable, they increment then return;
// after the variable they return then increment.
System.out.println(i++); // i = 1, prints 0 (post-increment)
System.out.println(++i); // i = 2, prints 2 (pre-increment)
System.out.println(i--); // i = 1, prints 2 (post-decrement)
System.out.println(--i); // i = 0, prints 0 (pre-decrement)

///////////////////////////////////////
// Control Structures
///////////////////////////////////////
System.out.println("\n->Control Structures");

// If statements are c-like
int j = 10;
if (j == 10) {
    System.out.println("I get printed");
} else if (j > 10) {
    System.out.println("I don't");
} else {
    System.out.println("I also don't");
}

// While loop
int fooWhile = 0;
while(fooWhile < 100) {
    System.out.println(fooWhile);
    // Increment the counter
    // Iterated 100 times, fooWhile 0,1,2...99
    fooWhile++;
}
System.out.println("fooWhile Value: " + fooWhile);

// Do While Loop
int fooDoWhile = 0;
do {
```

```
    System.out.println(fooDoWhile);
    // Increment the counter
    // Iterated 99 times, fooDoWhile 0->99
    fooDoWhile++;
} while(fooDoWhile < 100);
System.out.println("fooDoWhile Value: " + fooDoWhile);

// For Loop
// for loop structure => for(<start_statement>; <conditional>; <step>)
for (int fooFor = 0; fooFor < 10; fooFor++) {
    System.out.println(fooFor);
    // Iterated 10 times, fooFor 0->9
}
System.out.println("fooFor Value: " + fooFor);

// Nested For Loop Exit with Label
outer:
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; j++) {
    if (i == 5 && j ==5) {
      break outer;
      // breaks out of outer loop instead of only the inner one
    }
  }
}

// For Each Loop
// The for loop is also able to iterate over arrays as well as objects
// that implement the Iterable interface.
int[] fooList = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// for each loop structure => for (<object> : <iterable>)
// reads as: for each element in the iterable
// note: the object type must match the element type of the iterable.

for (int bar : fooList) {
    System.out.println(bar);
    //Iterates 9 times and prints 1-9 on new lines
}

// Switch Case
// A switch works with the byte, short, char, and int data types.
// It also works with enumerated types (discussed in Enum Types), the
// String class, and a few special classes that wrap primitive types:
// Character, Byte, Short, and Integer.
int month = 3;
String monthString;
switch (month) {
    case 1: monthString = "January";
            break;
    case 2: monthString = "February";
            break;
    case 3: monthString = "March";
            break;
    default: monthString = "Some other month";
```

```
            break;
}
System.out.println("Switch Case Result: " + monthString);

// Starting in Java 7 and above, switching Strings works like this:
String myAnswer = "maybe";
switch(myAnswer) {
    case "yes":
        System.out.println("You answered yes.");
        break;
    case "no":
        System.out.println("You answered no.");
        break;
    case "maybe":
        System.out.println("You answered maybe.");
        break;
    default:
        System.out.println("You answered " + myAnswer);
        break;
}

// Conditional Shorthand
// You can use the '?' operator for quick assignments or logic forks.
// Reads as "If (statement) is true, use <first value>, otherwise, use
// <second value>"
int foo = 5;
String bar = (foo < 10) ? "A" : "B";
System.out.println(bar); // Prints A, because the statement is true


//////////////////////////////////////
// Converting Data Types And Typecasting
//////////////////////////////////////

// Converting data

// Convert String To Integer
Integer.parseInt("123");//returns an integer version of "123"

// Convert Integer To String
Integer.toString(123);//returns a string version of 123

// For other conversions check out the following classes:
// Double
// Long
// String

// Typecasting
// You can also cast Java objects, there's a lot of details and deals
// with some more intermediate concepts. Feel free to check it out here:
// http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html


//////////////////////////////////////
```

```
        // Classes And Functions
        ////////////////////////////////////

        System.out.println("\n->Classes & Functions");

        // (definition of the Bicycle class follows)

        // Use new to instantiate a class
        Bicycle trek = new Bicycle();

        // Call object methods
        trek.speedUp(3); // You should always use setter and getter methods
        trek.setCadence(100);

        // toString returns this Object's string representation.
        System.out.println("trek info: " + trek.toString());

        // Double Brace Initialization
        // The Java Language has no syntax for how to create static Collections
        // in an easy way. Usually you end up in the following way:

        private static final Set<String> COUNTRIES = new HashSet<String>();
        static {
           validCodes.add("DENMARK");
           validCodes.add("SWEDEN");
           validCodes.add("FINLAND");
        }

        // But there's a nifty way to achieve the same thing in an
        // easier way, by using something that is called Double Brace
        // Initialization.

        private static final Set<String> COUNTRIES = new HashSet<String>() {{
            add("DENMARK");
            add("SWEDEN");
            add("FINLAND");
        }}

        // The first brace is creating a new AnonymousInnerClass and the
        // second one declares an instance initializer block. This block
        // is called when the anonymous inner class is created.
        // This does not only work for Collections, it works for all
        // non-final classes.

    } // End main method
} // End LearnJava class


// You can include other, non-public outer-level classes in a .java file,
// but it is good practice. Instead split classes into separate files.


// Class Declaration Syntax:
// <public/private/protected> class <class name> {
```

```
//    // data fields, constructors, functions all inside.
//    // functions are called as methods in Java.
// }

class Bicycle {

    // Bicycle's Fields/Variables
    public int cadence; // Public: Can be accessed from anywhere
    private int speed;  // Private: Only accessible from within the class
    protected int gear; // Protected: Accessible from the class and subclasses
    String name; // default: Only accessible from within this package

    static String className; // Static class variable

    // Static block
    // Java has no implementation of static constructors, but
    // has a static block that can be used to initialize class variables
    // (static variables).
    // This block will be called when the class is loaded.
    static {
        className = "Bicycle";
    }

    // Constructors are a way of creating classes
    // This is a constructor
    public Bicycle() {
        // You can also call another constructor:
        // this(1, 50, 5, "Bontrager");
        gear = 1;
        cadence = 50;
        speed = 5;
        name = "Bontrager";
    }

    // This is a constructor that takes arguments
    public Bicycle(int startCadence, int startSpeed, int startGear,
        String name) {
        this.gear = startGear;
        this.cadence = startCadence;
        this.speed = startSpeed;
        this.name = name;
    }

    // Method Syntax:
    // <public/private/protected> <return type> <function name>(<args>)

    // Java classes often implement getters and setters for their fields

    // Method declaration syntax:
    // <access modifier> <return type> <method name>(<args>)
    public int getCadence() {
        return cadence;
    }
```

```
    // void methods require no return statement
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

    public void slowDown(int decrement) {
        speed -= decrement;
    }

    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    //Method to display the attribute values of this Object.
    @Override // Inherited from the Object class.
    public String toString() {
        return "gear: " + gear + " cadence: " + cadence + " speed: " + speed +
            " name: " + name;
    }
} // end class Bicycle

// PennyFarthing is a subclass of Bicycle
class PennyFarthing extends Bicycle {
    // (Penny Farthings are those bicycles with the big front wheel.
    // They have no gears.)

    public PennyFarthing(int startCadence, int startSpeed) {
        // Call the parent constructor with super
        super(startCadence, startSpeed, 0, "PennyFarthing");
    }

    // You should mark a method you're overriding with an @annotation.
    // To learn more about what annotations are and their purpose check this
    // out: http://docs.oracle.com/javase/tutorial/java/annotations/
    @Override
    public void setGear(int gear) {
        gear = 0;
    }
}

// Interfaces
// Interface declaration syntax
```

```java
// <access-level> interface <interface-name> extends <super-interfaces> {
//     // Constants
//     // Method declarations
// }

// Example - Food:
public interface Edible {
    public void eat(); // Any class that implements this interface, must
                       // implement this method.
}

public interface Digestible {
    public void digest();
}


// We can now create a class that implements both of these interfaces.
public class Fruit implements Edible, Digestible {

    @Override
    public void eat() {
        // ...
    }

    @Override
    public void digest() {
        // ...
    }
}

// In Java, you can extend only one class, but you can implement many
// interfaces. For example:
public class ExampleClass extends ExampleClassParent implements InterfaceOne,
    InterfaceTwo {

    @Override
    public void InterfaceOneMethod() {
    }

    @Override
    public void InterfaceTwoMethod() {
    }

}

// Abstract Classes

// Abstract Class declaration syntax
// <access-level> abstract <abstract-class-name> extends <super-abstract-classes> {
//     // Constants and variables
//     // Method declarations
// }

// Marking a class as abstract means that it contains abstract methods that must
```

```java
// be defined in a child class. Similar to interfaces, abstract classes cannot
// be instantiated, but instead must be extended and the abstract methods
// defined. Different from interfaces, abstract classes can contain a mixture of
// concrete and abstract methods. Methods in an interface cannot have a body,
// unless the method is static, and variables are final by default, unlike an
// abstract class. Also abstract classes CAN have the "main" method.

public abstract class Animal
{
    public abstract void makeSound();

    // Method can have a body
    public void eat()
    {
        System.out.println("I am an animal and I am Eating.");
        // Note: We can access private variable here.
        age = 30;
    }

    // No need to initialize, however in an interface
    // a variable is implicitly final and hence has
    // to be initialized.
    protected int age;

    public void printAge()
    {
        System.out.println(age);
    }

    // Abstract classes can have main function.
    public static void main(String[] args)
    {
        System.out.println("I am abstract");
    }
}

class Dog extends Animal
{
    // Note still have to override the abstract methods in the
    // abstract class.
    @Override
    public void makeSound()
    {
        System.out.println("Bark");
        // age = 30;    ==> ERROR!  age is private to Animal
    }

    // NOTE: You will get an error if you used the
    // @Override annotation here, since java doesn't allow
    // overriding of static methods.
    // What is happening here is called METHOD HIDING.
    // Check out this awesome SO post: http://stackoverflow.com/questions/16313649/
    public static void main(String[] args)
    {
```

```
        Dog pluto = new Dog();
        pluto.makeSound();
        pluto.eat();
        pluto.printAge();
    }
}

// Final Classes

// Final Class declaration syntax
// <access-level> final <final-class-name> {
//     // Constants and variables
//     // Method declarations
// }

// Final classes are classes that cannot be inherited from and are therefore a
// final child. In a way, final classes are the opposite of abstract classes
// because abstract classes must be extended, but final classes cannot be
// extended.
public final class SaberToothedCat extends Animal
{
    // Note still have to override the abstract methods in the
    // abstract class.
    @Override
    public void makeSound()
    {
        System.out.println("Roar");
    }
}

// Final Methods
public abstract class Mammal()
{
    // Final Method Syntax:
    // <access modifier> final <return type> <function name>(<args>)

    // Final methods, like, final classes cannot be overridden by a child class,
    // and are therefore the final implementation of the method.
    public final boolean isWarmBlooded()
    {
        return true;
    }
}


// Enum Type
//
// An enum type is a special data type that enables for a variable to be a set
// of predefined constants. The variable must be equal to one of the values that
// have been predefined for it. Because they are constants, the names of an enum
// type's fields are in uppercase letters. In the Java programming language, you
// define an enum type by using the enum keyword. For example, you would specify
// a days-of-the-week enum type as:
```

```java
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

// We can use our enum Day like that:

public class EnumTest {

    // Variable Enum
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY:
            case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs(); // => Mondays are bad.
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs(); // => Midweek days are so-so.
    }
}

// Enum types are much more powerful than we show above.
// The enum body can include methods and other fields.
// You can se more at https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
```

## Further Reading

The links provided here below are just to get an understanding of the topic, feel free to Google and find specific examples.

**Official Oracle Guides**:

- Java Tutorial Trail from Sun / Oracle
- Java Access level modifiers
- Object-Oriented Programming Concepts:
  - Inheritance
  - Polymorphism
  - Abstraction
- Exceptions
- Interfaces
- Generics
- Java Code Conventions

**Online Practice and Tutorials**

- Learneroo.com - Learn Java
- Codingbat.com

**Books**:

- Head First Java
- Thinking in Java
- Objects First with Java
- Java The Complete Reference