# Java coding language

Java is known for being simple, portable, secure, and robust.

Java Virtual Machine, which ensures the same Java code can be run on different operating systems and platforms. Sun Microsystems' slogan for Java was "write once, run everywhere".

**Note:** Java files have a .java extension.

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

System is a built-in Java class that contains useful tools for our programs.

out is short for "output".

println is short for "print line".

System.out.print() prints all statements on one line there are no new line.

## Comments in Java

Short comments use the single-line syntax: //

Multi line comments use the multi-line syntax: /* and */.

Another type of commenting option is the Javadoc comment which is represented by /** and */. Javadoc comments are used to create documentation for APIs.

Compile the java file in the terminal by typing (javac filename.java)

Java is a case-sensitive language. Case sensitivity means that syntax, the words our computer understands, must match the case.

## Data types

- Int:  used for store whole number (positive numbers, negative numbers, and zero) (values between -2,147,483,648 and 2,147,483,647)
- Double: hold decimals as well as very large and very small numbers. (The maximum value is 1.797,693,134,862,315,7 E+308. The minimum value is 4.9 E-324,)
- Boolean: hold one of two values: true or false.
- Char: hold only one character. (It must be surrounded by single quotes, ')
- String: hold sequences of characters. enclosed in double-quotes ("").
  There are three escape sequences to be aware of for string:
  1. The \" escape sequence allows us to add quotation marks " to a String value.
  2. Using the \\ escape sequence allows us to place backslashes in our String text.
  3. place a \n escape sequence in a String, the compiler will output a new line of text.

**Note:** Variable names of only one word are spelled in all lowercase letters. Variable names of more than one word have the first letter lowercase while the beginning letter of each subsequent word is capitalized. This style of capitalization is called camelCase.

**Note:** A variable starts with a valid letter, or a $, or a _. No other symbols or numbers can begin with a variable name. 1stPlace and *Gazer are not valid variable names.

## Operations Java

The order of operations: parentheses -> exponents -> multiplication, division, modulo -> addition, subtraction

Compound assignment operators perform an arithmetic operation on a variable and then reassign its value. Compound assignment operators for all of the arithmetic operators we've covered:

- Addition (+=)
- Subtraction (-=)
- Multiplication (*=)
- Division (/=)
- Modulo (%=)

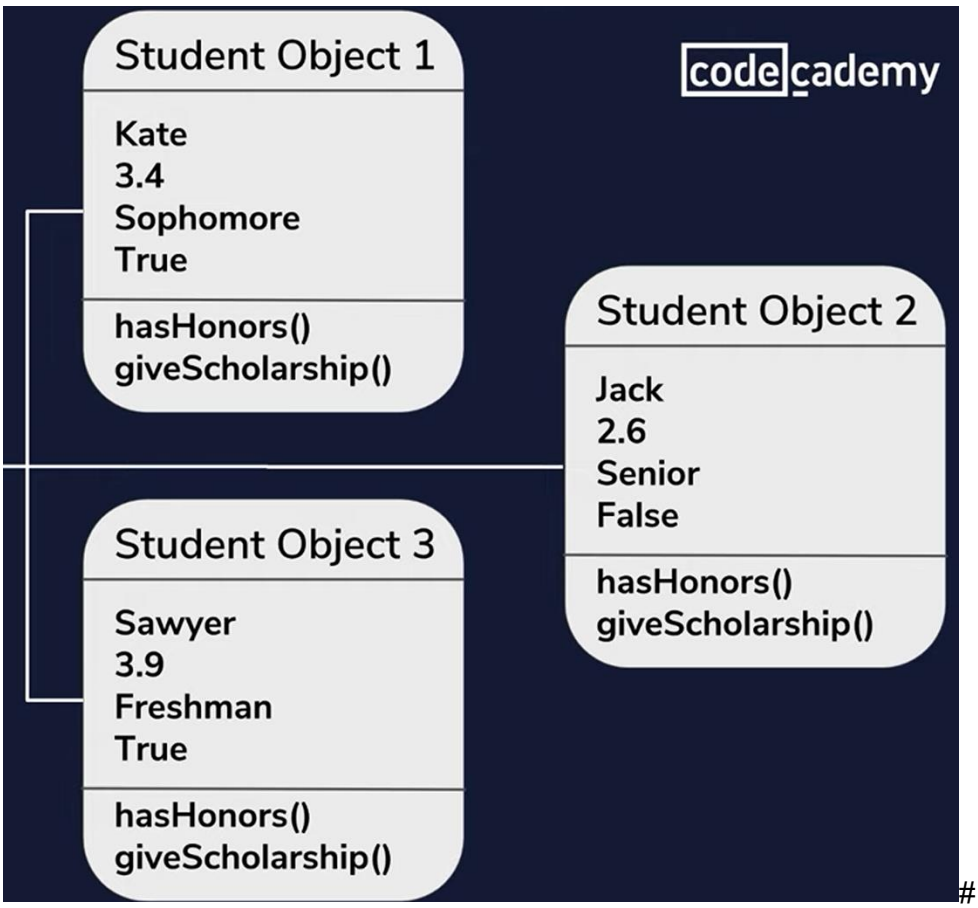
**Note:** To declare a variable with a value that cannot be manipulated, we need to use the final keyword.

```
final int yearBorn = 1968;
```

When we declare a variable using final, the value cannot be changed; any attempts at doing so will cause an error to occur

## Create data type

To create a new custom data type, we can use classes and identify the attribute and functions of the new data type.


Student Object 1
Kate
3.4
Sophomore
True
hasHonors()
giveScholarship()
codecademy
Student Object 2
Jack
2.6
Senior
False
hasHonors()
giveScholarship()
Student Object 3
Sawyer
3.9
Freshman
True
hasHonors()
giveScholarship()
#


Lightbulb
isOn (boolean)
watts (number)
turnOn() (func)
Microphone
color (string)
isAnalog (boolean)
record() (func)
Animal
color (string)
age (number)
speak() (func)
Date
month (number)
year (number)
toString() (func)

## Introduction To Classes

**A class:** is a template for creating objects in Java. A class outlines the necessary components and how they interact with each other. Example:

```
public class Car {
  // Empty Java Class
}
```

**Constructor:** is a special type of method defined within the class, used to initialize fields when an instance of the class is created.

**Note:** The name of the constructor method must be the same as the class itself.

```
public class Car {

 // Constructor
 public Car() {

   // instructions for creating a Car instance
 }
}
```

Create an instances of the constructor

```
Car ferrari = new Car();
```

If we print the value of the variable ferrari we would see its memory address: Car@76ed5528 In the above example, our variable ferrari is declared as a reference data type rather than with a primitive data type like int or boolean. This means that the variable holds a reference to the memory address of an instance. During its declaration, we specify the class name as the variable's type, which in this case is Car.

If we use a special value, null, we can initialize a reference-type variable without giving it a reference. If we were to assign null to an object, it would have a void reference because null has no value.

```
Car thunderBird = new Car();

System.out.println(thunderBird); // Prints:
Car@76ed5528

thunderBird = null; // change value to null

System.out.println(thunderBird); // Prints: null
```

**instance variables:** are often characterized by their “has-a” relationship with the object. each object created from the class will have its own copy of these variables.

```
public class Dog{

  public String name;
  public String breed;
  public int weight;
  public Dog(){
    //DO NOT WRITE ANYTHING HERE!!
  }
}
```

These fields can be set in the following three ways:

- If they are public, they can be set like this instanceName.fieldName = someValue;
- They can be set by class methods.
- They can be set by the constructor method

**Parameters:** are placeholders that we can use to pass information to a method.

```
public class Store{
  public String productType;
  public Store(String product){
    productType =  product;
  }
}
```

**Note:** A class can have multiple constructors. We can differentiate them based on their parameters. The signature helps the compiler to differentiate between different methods.

**Note:** An argument refers to the actual values passed during the method call while a parameter refers to the variables declared in the method signature.

**Methods:** are repeatable, modular blocks of code used to accomplish specific tasks.

In order to call a non-static method, we must call the method on the object we created.

```
Object myObject = new Object("red");
Object.methodName();
```

**Note:** code generally runs in a top-down order where code execution starts at the top of a program and ends at the bottom of a program; however, methods are ignored by the compiler unless they are being called.

**Note:** We mark the domain of this task using curly braces: {, and }. Everything inside the curly braces is part of the task. This domain is called the scope of a method.

**toString()** method can return a String that will print when we print the object

```
String object;
public String toString(){
      return "This is a " + object + " car!";
  }
```

When printing an instance of the constructor the result will be memory position.

```
//instance of the constructor
Store cookieShop = new Store("Cookies", 5);
//printing the instance
System.out.println(cookieShop);
//result: Store@7ad041f3
```

To make it print a useful text we use **toString()** method. When we define a **toString()** method for a class, we can return a String that will print when we print the object(instance).

## Conditionals and Control Flow

- If-Then-Else

```
if (True) {

  // Enroll in course

} else {

  // Enroll in prerequisite

}
```

- If-Then-Else-If

```
else if (course.equals("Theatre")) {

  // Enroll in Theatre course

}
```

**Note:** When we implement nested conditional statements, the outer statement is evaluated first. If the outer condition is true, then the inner, nested statement is evaluated.

- Switch Statement: check a given value against any number of conditions and run the code block where there is a match.

```
String course = "History";

switch (course) {
  case "Algebra":
    // Enroll in Algebra
    break;
  case "Biology":
    // Enroll in Biology
    break;
  case "History":
    // Enroll in History
    break;
  case "Theatre":
    // Enroll in Theatre
    break;
  default:
    System.out.println("Course not found");
}
```

## Conditional Operators

- AND operator (&&): used when multiple conditions are true.
- OR operator (||): used when at least one of two conditions are true.
- produce the opposite value, where true becomes false and false becomes true, with the NOT operator: !

### Combining Conditional Operators

```
boolean foo = true && !(false || !true)
```

The order of evaluation when it comes to conditional operators is as follows:

- Conditions placed in parentheses - ()
- NOT - !
- AND - &&
- OR - ||

## Java arrays

An array holds a fixed number of values of one type. Arrays hold doubles, ints, booleans, or any other primitives. Arrays can also contain Strings as well as object references!

Notice that the indexes start at 0!


```
vaildDtatType[] arrayName;
int[] nums;
String[] names;
```

```
double[] prices = {13.15, 15.87, 14.22, 16.66};
```

To create an array, we provide a name and declare the type of data it holds

Note: When we import a package in Java, we are making all of the methods of that package available in our code. We put this at the top of the file, before we even define the class!

```
import java.util.Arrays;
```

When we pass an array into Arrays.toString(), we can see the contents of the array printed out

```
  public static void main(String[] args){
    int[] lotteryNumbers = {4, 8, 15, 16, 23,
42};
    String betterPrintout
= Arrays.toString(lotteryNumbers);
    System.out.println(betterPrintout);
  }
```

Get Element By Index: We use square brackets, [ and ], to access data at a certain index

```
System.out.println(prices[1]);
```

**Note:** If we try to access an element outside of its appropriate index range, we will receive an ArrayIndexOutOfBoundsException error.

Creating an Empty Array

Empty arrays have to be initialized with a fixed size and Once you declare this size, it cannot be changed!

```
String[] menuItems = new String[5];
```

```
//getting how many objects in the array
arrayName.length;
```

**String[] args**

A String[] is an array made up of Strings

The args parameter is another example of a String array. In this case, the array args contains the arguments that we pass in from the terminal when we run the class file.

When we run the file HelloYou in the terminal with an argument of "Laura"

```
java HelloYou Laura
```

```
public class HelloYou {
  public static void main(String[] args) {
    System.out.println("Hello " + args[0]);
  }
}
```

We get the output:

```
Hello Laura
```

The String[] args would be interpreted as an array with one element, "Laura".

## ArrayLists

Used to create mutable and dynamic lists. It allows us to:

- Store object references as elements
- Store elements of the same type (just like arrays)
- Access elements by index (just like arrays)
- Add elements
- Remove elements

```
import java.util.ArrayList;
```

Creating ArrayLists

```
ArrayList<dataType> arrayListName;
```

We use angle brackets < and > to declare the type of the ArrayList. These symbols are used for generics. Generics are a Java construct that allows us to define classes and objects as parameters of an ArrayList. For this reason, we can't use primitive types in an ArrayList:

```
// This code won't compile:
ArrayList<int> ages;

// This code will compile:
ArrayList<Integer> ages;
```

ArrayList data type: <Integer> instead of int, <Double>, <Character> instead of char

```
//add an element to the end of the arrayList
arrayListName.add(object);
//add an element at a specific index of ArrayList
arrayListName.add(indexNum,object);
arrayListName.add(1,object);
```

```
// Declaring:
ArrayList<Integer> ages;
// Initializing:
ages = new ArrayList<Integer>();
```

| ArrayList Method Name | Purpose |
|---|---|
| add() | Adding a new ArrayList item. |
| size() | Accessing the size of an ArrayList. |
| get() | Finding an item by index. |
| set() | Changing element value from an Arraylist. it take two parameters index number and new value. |
| remove() | Removing an item with a specific value (index number or element value). |
| indexOf() | Retrieving the index of an item with a specific value. |
| addAll | Used to convert an array to arrayList<br>`arrayListName..addAll(Arrays.asList(arrayName));` |

**Note:** By inserting a value at a specified index, any elements that appear after this new element will have their index value shift over by 1.

**Note**: that an error will occur if we try to insert a value at an index that does not exist.

it is possible to create an ArrayList that holds values of different types. Example assortment is an ArrayList that can store different values because we do not specify its type during initialization.

```
ArrayList assortment = new ArrayList<>();
assortment.add("Hello"); // String
assortment.add(12); // Integer
assortment.add(ferrari); // reference to Car
// assortment holds ["Hello", 12, ferrari]
```

## Introduction to Loops

A loop is a programming tool that allows developers to repeat the same block of code until some condition is met. three types of loops:

1. while loops

```
while (silliness > 10) {

  // code to run

}
```

Using counter

```
// counter is initialized
int wishes = 0;

// conditional logic uses counter
while (wishes < 3) {

  System.out.println("Wish granted.");
  // counter is incremented
  wishes++;

}
```

2. for loops
   made up of the following three parts, each separated by a semicolon:
   - The initialization of the loop control variable.
   - A boolean expression.
   - An increment or decrement statement.

```
for (int i = 0; i < 5; i++) {

  // code that will run

}
```

3. for-each loops
   llow us to directly loop through each item in a list of items (like an array or ArrayList) and perform some action with each item.

```
for (String inventoryItem : inventoryItems) {
  // Print element value
  System.out.println(inventoryItem);

}
```

**Note:** We can name the enhanced for loop variable whatever we want; using the singular of a plural is just a convention. We may also encounter conventions like String word : sentence.

**Note:** The break keyword is used to exit, or break, a loop. Once break is executed, the loop will stop iterating.

**Note:** The continue keyword can be placed inside of a loop if we want to skip an iteration. If continue is executed, the current loop iteration will immediately end, and the next iteration will begin.

**Note:** If the return keyword was executed inside a loop contained in a method, then the loop iteration would be stopped and the method/constructor would be exited.

Removing Elements During Traversal

When using a while loop and removing elements from an ArrayList, we should not increment the while loop's counter whenever we remove an element. We don't need to increase the counter because all of the other elements have now shifted to the left.

When using a for loop, we, unfortunately, must increase our loop control variable — the loop control variable will always change when we reach the end of the loop. Since we can't avoid increasing our loop control variable, we can take matters into our own hands and decrease the loop control variable whenever we remove an item.

## String Methods

| String Methods | Purpose |
|---|---|
| length() | returns the length — total number of characters — of a String.<br>`String str = "Hello World!";`<br>`System.out.println(str.length());` |
| concat() | concatenates one string to the end of another string.<br>`String name = new String("Code");`<br>`name = name.concat("cademy");`<br>`System.out.println(name);` |
| equals() | test equality with strings. Return true or false.<br>`System.out.println(String1.equals(String2));` |
| equalsIgnoreCase() | compares two strings without considering upper/lower cases. Return true or false. |
| compareTo() | each character in the String is converted to Unicode; then the Unicode character from each String is compared. return an int that represents the difference between the two Strings.<br>`System.out.println(String1.compareTo(String2));` |
| indexOf() | index of the first occurrence of a character in a string,<br>`System.out.println(letters.indexOf("C"));`<br>**Note:** If the indexOf() doesn't find what it's looking for, it'll return a -1. |
| charAt() | returns the character located at a String's specified index.<br>`String str = "qwer";`<br>`System.out.println(str.charAt(2));` |

| | |
|---|---|
| substring() | extract a substring from a string. called with two arguments, the first argument is inclusive and the second is exclusive.<br>`line.substring(27, 33)` |
| toUpperCase() | returns the string value converted to uppercase. |
| toLowerCase() | returns the string value converted to lowercase. |

## Access, Encapsulation and Scope

### Access

The public and private defining what parts of your code have access to other parts of your code. And it can be defined for variables, methods, constructors, and even a class.

**public** means that any part of our code can interact with variables, methods, constructors, or class - even if that code is in a different class!

**Note:** While you can set classes and constructors to private, it's fairly uncommon to do so.

When a Class' instance variable or method is marked as private, that means that you can only access those structures from elsewhere inside that same class.

**Note:** By limiting access by using the private keyword, we are able to segment, or encapsulate, our code into individual units.

In classes, we often make all of our instance variables private. And create accessor method (getter) to give other classes access to a private instance variable.

```
public class Dog{
  private String name;

  //Other methods and constructors

  public String getName() {
    return name;
  }
}
```

Accessor methods will always be public, and will have a return type that matches the type of the instance variable they're accessing.

mutator methods ( setters). These methods allow other classes to reset the value stored in private instance variables.

Mutator methods often are void methods — they don't return anything, they often have one parameter that is the same type as the variable.

```
private String name;

//Other methods and constructors

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

## Scope

Local variables: define within method, Loop or statement.

Instance variables: define at the start of the class.

instance variables are declared inside a class but outside any methods or constructors, all methods and constructors are within the scope of that variable.

**Note:** if there are instance variables and method has parameter with the same name, Java refers to the local variable name. so if the variable used in the method it will be the parameter but If we wanted to access the instance variable and not the local variable, we could use the this keyword.

The **this** keyword is a reference to the current object.

When Using this With Methods it mean that the same object used the method will be used to call instead of this.

```
public String name;
public Dog(String name){
  this.name = name;
}
```

```
public class Computer{
  public int brightness;
  public int volume;

  public void setBrightness(int inputBrightness){
    this.brightness = inputBrightness;
  }

  public void setVolume(int inputVolume){
    this.volume = inputVolume;
  }

  public void resetSettings(){
    this.setBrightness(0);
    this.setVolume(0);
  }
}
```

```
public static void main(String[] args){
  Computer myComputer = new Computer();
  myComputer.resetSettings();
}
```

In this example, calling myComputer.resetSettings() is as if we called myComputer.setBrightness(0) and myComputer.setVolume(0). this serves as a placeholder for whatever object was used to call the original method.

Encapsulation is a technique used to keep implementation details hidden from other classes. Its aim is to create small bundles of logic.

The **this** keyword can be used to designate the difference between instance variables and local variables.

## Math Class

| Math Methods | Purpose |
|---|---|
| abs(int x) | Returns the absolute value of an int value. The absolute value states how many numbers a value is away from 0.<br>`System.out.println(Math.abs(5)); // Prints: 5` |
| abs(double x) | Returns the absolute value of a double value.<br>`System.out.println(Math.abs(5.0)); // Prints: 5.0` |
| pow(double base, double exponent) | Returns the value of the first parameter raised to the power of the second parameter.<br>`double x = Math.pow(5, 3);` |
| sqrt(double x) | Returns the positive square root of a double value.<br>`double x = Math.sqrt(49);` |
| random() | Returns a double value greater than or equal to 0.0 and less than 1.0.<br>`System.out.println(Math.random());`<br>`// Random double value between 0 and 10, not including 10`<br>`double a = Math.random() * 10;`<br>`// Random int value between 0 and 9`<br>`int b = (int)(Math.random() * 10);`<br>`// Random int value between 10 and 20`<br>`int d = (int)(Math.random() * 11 ) + 10;` |

## Static Variables and Methods

**Static methods** are methods that belong to an entire class, not a specific object of the class. **Static methods** are called using the class name and the '**.**'operator.

**Note:** **Non_Static** method call by creating object (like Math myMathObject = new Math()) than use the object to call the method example myMathObject.max();

**static variables** belonging to the class itself instead of belonging to a particular object of the class. static variables can be accessed by using the name of the class and the '**.**' operator.

```
// Static variables
public static String genus = "Canis";
```

```
//Instance variables
public int age;
public String name;
```

**final** means that you can't change the variable's value after creating it.

**Note:** Unlike static methods, you can still access static variables from a specific object of the class. However, no matter what object you use to access the variable, the value will always be the same.

```
public static void myFirstStaticMethod(){
  // Some code here
}
```

**Note:** static methods can't interact with non-static instance variables.

**Note:** The **this** keyword can't be used by a static method since static methods are associated with an entire class, not a specific object of that class. If you try to mix this with a static method, you'll see the error message (non-static variable this cannot be referenced from a static context).

**Note:** Static methods and variables are associated with the class as a whole, not objects of the class.

**Note:** Both static methods and non-static methods can interact with static variables.

## Introducing Inheritance

**Inheritance** is an object-oriented programming (OOP) concept by which the properties and behaviors from the parent class are passed on to the child class.

- Parent class, superclass, and base class refer to the class that another class inherits from.
- Child class, subclass, and derived class refer to a class that inherits from another class.

**Note:** When we use inheritance to extend a subclass from a superclass, we create an "is-a" relationship from the subclass to the superclass.

```
class Parent {

  // Parent class members

}
/*the keyword extends define a
child class so that it
inherits from a parent class*/
class Child extends Parent {

  // additional Child class members

}
```

### Inheriting the Constructor

super() method which acts like the parent constructor inside the child class constructor.

write a constructor without making a call to any super() constructor.

```
class Triangle extends Shape {

  Triangle() {
    super(3);
    /*mkae the value for numSide
    3 for the child(Triangle) class*/
  }
  // additional Triangle class members
}
```

```
class Triangle extends Shape {

  Triangle() {
    this.numSides = 3;
    /*NOTE numSides is instance
    variable defined in the parent class*/
  }
  // additional Triangle class methods
}
```

**Protected** access modifier used to keep a parent class member accessible to its child classes and to files in the package it's contained in — and otherwise private.

| Modifier | Class | Package | Child Class | Global |
|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | ✓ |
| protected | ✓ | ✓ | ✓ | ✗ |
| no modifier | ✓ | ✓ | ✗ | ✗ |
| private | ✓ | ✗ | ✗ | ✗ |

**Note:** If we add final after a parent class method's access modifier, we disallow any child classes from changing that method.

## Introducing Polymorphism

**Polymorphism**, which derives from Greek meaning "many forms", allows a child class to share the information and behavior of its parent class while also incorporating its own functionality.

The main advantages of polymorphic programming:

- Simplifying syntax
- Reducing cognitive overload for developers

**Note:** that the reverse situation is not true; you cannot use a generic parent class instance where a child class instance is required. So, an Orange can be used as a Fruit, but a Fruit cannot be used as an Orange.

Notice that in order to properly override a child class method has the following in common with the corresponding method in parent class:

- Method name
- Return type
- Number and type of parameters

**@Override keyword** annotation informs the compiler that we want to override a method in the parent class.

fact, we can put instances of different classes that share a parent class together in an array or ArrayList!

```
Monster dracula, wolfman, zombie1;

dracula = new Vampire();
wolfman = new Werewolf();
zombie1 = new Zombie();

Monster[] monsters = {dracula, wolfman, zombie1};
```

```
for (Monster monster : monsters) {

  monster.attack();

}
```

**Note:** Each Java class requires its own file, but only one class in a Java package needs a main() method.

Java's OOP principle of **polymorphism** means you can use a child class object like a member of its parent class, but also give it its own traits.