https://www.codecademy.com/learn/learn-java/modules/learn-java-hello-world/cheatsheet

Table of Contents

Hello World

Print Line

```
System.out.println("Hello, world!");
// Output: Hello, world!
```

Comments

```
// I am a single line comment!

/*
And I am a
multi-line comment!
*/
```

Compiling Java

```
# Compile the class file:
javac hello.java
```

```
# Execute the compiled file:
java hello
```

main() methood

```
public class Person {

  public static void main(String[] args) {

    System.out.println("Hello, world!");

  }

}
```

Classes

```
public class Person {

  public static void main(String[] args) {

    System.out.println("I am a person, not a computer.");

  }

}
```

Variables

boolean

```
boolean result = true;
boolean isMarried = false;
```

Strings

```
String name = "Bob";

// The following will print "false" because strings are case-sensitive
System.out.println(name.equals("bob"));
```

Primitive Data Types

```
int age = 28;

char grade = 'A';

boolean late = true;

byte b = 20;

long num1 = 1234567;

short no = 10;

float k = (float)12.5;

double pi = 3.14;
```

Static Typing

```
int i = 10;          // type is int
char ch = 'a';       // type is char

j = 20;              // won't compile, no type is given
char name = "Lil";   // won't compile, wrong data type
```

final Keyword

```
// Value cannot be changed:
final double PI = 3.14;
```

double Data type

```
double PI = 3.14;
double price = 5.75;
```

Math Operations

```
int a = 20;
int b = 10;

int result;
```

```
result = a + b;  // 30

result = a - b;  // 10

result = a * b;  // 200

result = a / b;  // 2

result = a % b;  // 0
```

Comparison Operators

```
int a = 5;
int b = 3;

boolean result = a > b;
// result now holds the boolean value true
```

Compound Assignment Operators

```
int number = 5;

number += 3; // Value is now 8
number -= 4; // Value is now 4
number *= 6; // Value is now 24
number /= 2; // Value is now 12
number %= 7; // Value is now 5
```

Increment and Decrement Operators

```
int numApples = 5;
numApples++; // Value is now 6

int numOranges = 5;
numOranges--; // Value is now 4
```

Object Oriented Java

Java objects' state and behavior

```

```

```
public class Person {
  // state of an object
  int age;
  String name;

  // behavior of an object
  public void set_value() {
    age = 20;
    name = "Robin";
  }
  public void get_value() {
    System.out.println("Age is " + age);
    System.out.println("Name is " + name);
  }

  // main method
  public static void main(String [] args) {
    // creates a new Person object
    Person p = new Person();

    // changes state through behavior
    p.set_value();
  }
}
```

Java instance

```
public class Person {
  int age;
  String name;

  // Constructor method
  public Person(int age, String name) {
    this.age = age;
    this.name = name;
  }

  public static void main(String[] args) {
    Person Bob = new Person(31, "Bob");
    Person Alice = new Person(27, "Alice");
  }
}
```

Java dot notation

```
public class Person {
  int age;
```

```
  public static void main(String [] args) {
    Person p = new Person();

    // here we use dot notation to set age
    p.age = 20;

    // here we use dot notation to access age and print
    System.out.println("Age is " + p.age);
    // Output: Age is 20
  }
}
```

Constructor Method in Java

```
public class Maths {
  public Maths() {
    System.out.println("I am constructor");
  }
  public static void main(String [] args) {
    System.out.println("I am main");
    Maths obj1 = new Maths();
  }
}
```

Creating a new Class instance in Java

```
public class Person {
  int age;
  // Constructor:
  public Person(int a) {
    age = a;
  }

  public static void main(String [] args) {
    // Here, we create a new instance of the Person class:
    Person p = new Person(20);
    System.out.println("Age is " + p.age); // Prints: Age is 20
  }
}
```

Reference Data Types

```
public class Cat {
  public Cat() {
    // instructions for creating a Cat instance
```

```
  }

  public static void main(String[] args) {
    // garfield is declared with reference data type `Cat`
    Cat garfield = new Cat();
    System.out.println(garfield); // Prints: Cat@76ed5528
  }
}
```

Constructor Signatures

```
// The signature is `Cat(String furLength, boolean hasClaws)`.
public class Cat {
  String furType;
  boolean containsClaws;

  public Cat(String furLength, boolean hasClaws) {
    furType = furLength;
    containsClaws = hasClaws;
  }
  public static void main(String[] args) {
    Cat garfield = new Cat("Long-hair", true);
  }
}
```

null Values

```
public class Bear {
  String species;
  public Bear(String speciesOfBear;) {
    species = speciesOfBear;
  }

  public static void main(String[] args) {
    Bear baloo = new Bear("Sloth bear");
    System.out.println(baloo); // Prints: Bear@4517d9a3
    // set object to null
    baloo = null;
    System.out.println(baloo); // Prints: null
  }
}
```

The body of a Java method

```
public class Maths {
  public static void sum(int a, int b) { // Start of sum
```

```
    int result = a + b;
    System.out.println("Sum is " + result);
  } // End of sum


  public static void main(String [] args) {
    // Here, we call the sum method
    sum(10, 20);
    // Output: Sum is 30
  }
}
```

Method parameters in Java

```
public class Maths {
  public int sum(int a, int b) {
    int k = a + b;
    return k;
  }

  public static void main(String [] args) {
    Maths m = new Maths();
    int result = m.sum(10, 20);
    System.out.println("sum is " + result);
    // prints - sum is 30
  }
}
```

Java Variables Inside a Method

```
//For example, `i` and `j` variables are available in the `main` method
only:

public class Maths {
  public static void main(String [] args) {
    int i, j;
    System.out.println("These two variables are available in main method
only");
  }
}
```

Returning info from a Java method

```
public class Maths {

  // return type is int
```

```
  public int sum(int a, int b) {
    int k;
    k = a + b;

    // sum is returned using the return keyword
    return k;
  }

  public static void main(String [] args) {
    Maths m = new Maths();
    int result;
    result = m.sum(10, 20);
    System.out.println("Sum is " + result);
    // Output: Sum is 30
  }
}
```

Declaring a Method

```
// Here is a public method named sum whose return type is int and has two
int parameters a and b
public int sum(int a, int b) {
  return(a + b);
}
```

Conditionals and Control Flow

else Statement

```
boolean condition1 = false;

if (condition1){
    System.out.println("condition1 is true");
}
else{
    System.out.println("condition1 is not true");
}
// Prints: condition1 is not true
```

else if Statements

```
int testScore = 76;
char grade;

if (testScore >= 90) {
```

```
  grade = 'A';
} else if (testScore >= 80) {
  grade = 'B';
} else if (testScore >= 70) {
  grade = 'C';
} else if (testScore >= 60) {
  grade = 'D';
} else {
  grade = 'F';
}

System.out.println("Grade: " + grade); // Prints: C
```

if Statement

```
if (true) {
    System.out.println("This code executes");
}
// Prints: This code executes

if (false) {
    System.out.println("This code does not execute");
}
// There is no output for the above statement
```

Nested Conditional Statements

```
boolean studied = true;
boolean wellRested = true;

if (wellRested) {
  System.out.println("Best of luck today!");
  if (studied) {
    System.out.println("You are prepared for your exam!");
  } else {
    System.out.println("Study before your exam!");
  }
}

// Prints: Best of luck today!
// Prints: You are prepared for your exam!
```

NOT Operator

```
boolean a = true;
System.out.println(!a); // Prints: false
```

```
System.out.println(!false) // Prints: true
```

AND Operator

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

The OR Operator

```
System.out.println(true || true); // Prints: true
System.out.println(true || false); // Prints: true
System.out.println(false || true); // Prints: true
System.out.println(false || false); // Prints: false
```

Conditional Operators - Order of Evaluation

```
boolean foo = true && (!false || true); // true
/*
(!false || true) is evaluated first because it is contained within
parentheses.

Then !false is evaluated as true because it uses the NOT operator.

Next, (true || true) is evaluation as true.

Finally, true && true is evaluated as true meaning foo is true. */
```

Arrays and ArrayLists

Java ArrayList

```
// import the ArrayList package
import java.util.ArrayList;

// create an ArrayList called students
ArrayList<String> students = new ArrayList<String>();
```

Index

```
int[] marks = {50, 55, 60, 70, 80};

System.out.println(marks[0]);
// Output: 50

System.out.println(marks[4]);
// Output: 80
```

Arrays

```
// Create an array of 5 int elements
int[] marks = {10, 20, 30, 40, 50};
```

Array creation in Java

```
int[] age = {20, 21, 30};

int[] marks = new int[3];
marks[0] = 50;
marks[1] = 70;
marks[2] = 93;
```

Changing an Element Value

```
int[] nums = {1, 2, 0, 4};
// Change value at index 2
nums[2] = 3;
```

Modifying ArrayLists in Java

```
import java.util.ArrayList;

public class Students {
  public static void main(String[] args) {

     // create an ArrayList called studentList, which initially holds []
        ArrayList<String> studentList = new ArrayList<String>();

    // add students to the ArrayList
```

```
    studentList.add("John");
    studentList.add("Lily");
    studentList.add("Samantha");
    studentList.add("Tony");

    // remove John from the ArrayList, then Lily
    studentList.remove(0);
    studentList.remove("Lily");

    // studentList now holds [Samantha, Tony]

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

Loops

For-each statement in Java

```
// array of numbers
int[] numbers = {1, 2, 3, 4, 5};

// for-each loop that prints each number in numbers
// int num is the handle while numbers is the source array
for (int num : numbers) {
    System.out.println(num);
}
```

String Methods

length() String Method in Java

```
String str = "Codecademy";

System.out.println(str.length());
// prints 10
```

indexOf() String Method in Java

```
String str = "Hello World!";

System.out.println(str.indexOf("l"));
// prints 2
```

```
System.out.println(str.indexOf("Wor"));
// prints 6

System.out.println(str.indexOf("z"));
// prints -1
```

concat() String Method in Java

```
String s1 = "Hello";
String s2 = " World!";

String s3 = s1.concat(s2);
// concatenates strings s1 and s2

System.out.println(s3);
// prints "Hello World!"
```

String Method equals() in Java

```
String s1 = "Hello";
String s2 = "World";

System.out.println(s1.equals("Hello"));
// prints true

System.out.println(s2.equals("Hello"));
// prints false

System.out.println(s2.equalsIgnoreCase("world"));
// prints true
```

charAt() String Method in Java

```
String str = "This is a string";

System.out.println(str.charAt(0));
// prints 'T'

System.out.println(str.charAt(15));
// prints 'g'
```

toUpperCase() and toLowerCase() String Methods

```

```

```
String str = "Hello World!";

String uppercase = str.toUpperCase();
// uppercase = "HELLO WORLD!"

String lowercase = str.toLowerCase();
// lowercase = "hello world!"
```

Access, Encapsulation, and Static Methods

The static Keyword

Static methods and variables are declared as static by using the static keyword upon declaration.

```
public class ATM {
  // Static variables
  public static int totalMoney = 0;
  public static int numATMs = 0;

  // A static method
  public static void averageMoney(){
    System.out.println(totalMoney / numATMs);
  }
  }
```

Static Methods and Variables

Static methods and variables are associated with the class as a whole, not objects of the class. Both are used by using the name of the class followed by the . operator.

```
public class ATM{
  // Static variables
  public static int totalMoney = 0;
  public static int numATMs = 0;

  // A static method
  public static void averageMoney(){
    System.out.println(totalMoney / numATMs);
  }

  public static void main(String[] args){

    //Accessing a static variable
    System.out.println("Total number of ATMs: " + ATM.numATMs);

    // Calling a static method
```

```
    ATM.averageMoney();
  }

}
```

Static Methods with Instance Variables

Static methods cannot access or change the values of instance variables.

```
class ATM{
// Static variables
  public static int totalMoney = 0;
  public static int numATMs = 0;

  public int money = 1;

  // A static method
  public static void averageMoney(){
    // Can not use this.money here because a static method can't access
instance variables
  }

}
```

Methods with Static Variables

Both non-static and static methods can access or change the values of static variables.

```
class ATM{
// Static variables
  public static int totalMoney = 0;
  public static int numATMs = 0;
  public int money = 1;

 // A static method interacting with a static variable
  public static void staticMethod(){
    totalMoney += 1;
   }

  // A non-static method interactingwith a static variable
  public void nonStaticMethod(){
    totalMoney += 1;
  }
}
```

Static Methods and the this Keyword

Static methods do not have a this reference and are therefore unable to use the class's instance variables or call non-static methods.

```
public class DemoClass{

  public int demoVariable = 5;

  public void demoNonStaticMethod(){

  }
  public static void demoStaticMethod(){
    // Can't use "this.demoVariable" or "this.demoNonStaticMethod()"
  }
}
```

The public and private keywords

In Java, the keywords public and private define the access of classes, instance variables, constructors, and methods.

private restricts access to only the class that declared the structure, while public allows for access from any class.

Encapsulation

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

The private Keyword

In Java, instance variables are encapsulated by using the private keyword. This prevents other classes from directly accessing these variables.

```
public class CheckingAccount{
  // Three private instance variables
  private String name;
  private int balance;
  private String id;
}
```

Accessor Methods

In Java, accessor methods return the value of a private variable. This gives other classes access to that value stored in that variable. without having direct access to the variable itself.

Accessor methods take no parameters and have a return type that matches the type of the variable they are accessing.

```
public class CheckingAccount{
  private int balance;

  //An accessor method
  public int getBalance(){
    return this.balance;
  }
}
```

Mutator Methods

In Java, mutator methods reset the value of a private variable. This gives other classes the ability to modify the value stored in that variable without having direct access to the variable itself.

Mutator methods take one parameter whose type matches the type of the variable it is modifying. Mutator methods usually don't return anything.

```
public class CheckingAccount{
  private int balance;

  //A mutator method
  public void setBalance(int newBalance){
    this.balance = newBalance;
  }
}
```

Local Variables

In Java, local variables can only be used within the scope that they were defined in. This scope is often defined by a set of curly brackets. Variables can't be used outside of those brackets.

```
public void exampleMethod(int exampleVariable){
  // exampleVariable can only be used inside these curly brackets.
}
```

The this Keyword with Variables

In Java, the this keyword can be used to designate the difference between instance variables and local variables. Variables with this. reference an instance variable.

```
public class Dog{
  public String name;

  public void speak(String name){
    // Prints the instance variable named name
    System.out.println(this.name);

    // Prints the local variable named name
    System.out.println(name);
  }
}
```

The this Keyword with Methods

In Java, the this keyword can be used to call methods when writing classes.

```
public class ExampleClass{
  public void exampleMethodOne(){
    System.out.println("Hello");
  }

  public void exampleMethodTwo(){
    //Calling a method using this.
    this.exampleMethodOne();
    System.out.println("There");
  }
}
```

Static Methods

Static methods are methods that can be called within a program without creating an object of the class.

```
// static method
public static int getTotal(int a, int b) {
  return a + b;
}

public static void main(String[] args) {
  int x = 3;
  int y = 2;
  System.out.println(getTotal(x,y)); // Prints: 5
}
```

Calling a Static Method

Static methods can be called by appending the dot operator to a class name followed by the name of the method.

```
int largerNumber = Math.max(3, 10); // Call static method
System.out.println(largerNumber); // Prints: 10
```

The Math Class

The Math class (which is part of the java.lang package) contains a variety of static methods that can be used to perform numerical calculations.

```
System.out.println(Math.abs(-7.0)); // Prints: 7

System.out.println(Math.pow(5, 3)); // Prints: 125.0

System.out.println(Math.sqrt(52)); // Prints: 7.211102550927978
```

Inheritance and Polymorphism

Inheritance in Java

Inheritance is an important feature of object-oriented programming in Java. It allows for one class (child class) to inherit the fields and methods of another class (parent class). For instance, we might want a child class Dog to inherent traits from a more general parent class Animal.

When defining a child class in Java, we use the keyword extends to inherit from a parent class.

```
// Parent Class
class Animal {
  // Animal class members
}

// Child Class
class Dog extends Animal {
  // Dog inherits traits from Animal

  // additional Dog class members
}
```

Main() method in Java

In simple Java programs, you may work with just one class and one file. However, as your programs become more complex you will work with multiple classes, each of which requires its own file. Only one of these files in the Java package requires a main() method, and this is the file that will be run in the package.

For example, say we have two files in our Java package for two different classes:

Shape, the parent class. Square, the child class. If the Java file containing our Shape class is the only one with a main() method, this is the file that will be run for our Java package.

```
// Shape.java file
class Shape {
  public static void main(String[] args) {
    Square sq = new Square();
  }
}

// Square.java file
class Square extends Shape {

}
```

super() in Java

In Java, a child class inherits its parent's fields and methods, meaning it also inherits the parent's constructor. Sometimes we may want to modify the constructor, in which case we can use the super() method, which acts like the parent constructor inside the child class constructor.

Alternatively, we can also completely override a parent class constructor by writing a new constructor for the child class.

```
// Parent class
class Animal {
  String sound;
  Animal(String snd) {
    this.sound = snd;
  }
}

// Child class
class Dog extends Animal {
  // super() method can act like the parent constructor inside the child
class constructor.
  Dog() {
    super("woof");
  }
  // alternatively, we can override the constructor completely by defining
a new constructor.
  Dog() {
    this.sound = "woof";
```

```
  }
}
```

Protected and Final keywords in Java

When creating classes in Java, sometimes we may want to control child class access to parent class members. We can use the protected and final keywords to do just that.

protected keeps a parent class member accessible to its child classes, to files within its own package, and by subclasses of this class in another package.

Adding final before a parent class method's access modifier makes it so that any child classes cannot modify that method - it is immutable.

```
class Student {
  protected double gpa;
  // any child class of Student can access gpa

  final protected boolean isStudent() {
    return true;
  }
  // any child class of Student cannot modify isStudent()
}
```

Polymorphism in Java

Java incorporates the object-oriented programming principle of polymorphism.

Polymorphism allows a child class to share the information and behavior of its parent class while also incorporating its own functionality. This allows for the benefits of simplified syntax and reduced cognitive overload for developers.

```
// Parent class
class Animal {
  public void greeting() {
    System.out.println("The animal greets you.");
  }
}

// Child class
class Cat extends Animal {
  public void greeting() {
    System.out.println("The cat meows.");
  }
}

class MainClass {
```

```
  public static void main(String[] args) {
    Animal animal1 = new Animal();  // Animal object
    Animal cat1 = new Cat();  // Cat object
    animal1.greeting(); // prints "The animal greets you."
    cat1.greeting(); // prints "The cat meows."
  }
}
```

Method Overriding in Java

In Java, we can easily override parent class methods in a child class. Overriding a method is useful when we want our child class method to have the same name as a parent class method but behave a bit differently.

In order to override a parent class method in a child class, we need to make sure that the child class method has the following in common with its parent class method:

Method name Return type Number and type of parameters Additionally, we should include the @Override keyword above our child class method to indicate to the compiler that we want to override a method in the parent class.

```
// Parent class
class Animal {
  public void eating() {
    System.out.println("The animal is eating.");
  }
}

// Child class
class Dog extends Animal {
  // Dog's eating method overrides Animal's eating method
    @Override
  public void eating() {
    System.out.println("The dog is eating.");
  }
}
```

Child Classes in Arrays and ArrayLists

In Java, polymorphism allows us to put instances of different classes that share a parent class together in an array or ArrayList.

For example, if we have an Animal parent class with child classes Cat, Dog, and Pig we can set up an array with instances of each animal and then iterate through the list of animals to perform the same action on each.

```
// Animal parent class with child classes Cat, Dog, and Pig.
Animal cat1, dog1, pig1;
```

```
cat1 = new Cat();
dog1 = new Dog();
pig1 = new Pig();

// Set up an array with instances of each animal
Animal[] animals = {cat1, dog1, pig1};

// Iterate through the list of animals and perform the same action with
each
for (Animal animal : animals) {

  animal.sound();

}
```

Two-Dimensional Arrays

Nested Iteration Statements

```
for(int outer = 0; outer < 3; outer++){
    System.out.println("The outer index is: " + outer);
    for(int inner = 0; inner < 4; inner++){
        System.out.println("\tThe inner index is: " + inner);
    }
}
```

Declaring 2D Arrays

```
int[][] twoDIntArray;
String[][] twoDStringArray;
double[][] twoDDoubleArray;
```

Accessing 2D Array Elements

```
//Given a 2d array called `arr` which stores `int` values
int[][] arr = { {1,2,3},
                {4,5,6}};

//We can get the value `4` by using
int retrieved = arr[1][0];
```

Initializer Lists

```
// Method one: declaring and intitializing at the same time
double[][] doubleValues = { {1.5, 2.6, 3.7}, {7.5, 6.4, 5.3}, {9.8,  8.7,
7.6}, {3.6, 5.7, 7.8}};

// Method two: declaring and initializing separately:
String[][] stringValues;
stringValues = new String[][] { {"working", "with"}, {"2D", "arrays"},
{"is", "fun"}};
```

Modify 2D Array Elements

```
double[][] doubleValues = { {1.5, 2.6, 3.7}, {7.5, 6.4, 5.3}, {9.8,  8.7,
7.6}, {3.6, 5.7, 7.8}};

doubleValues[2][2] = 100.5;
// This will change the value 7.6 to 100.5
```

Row-Major Order

```
for(int i = 0; i < matrix.length; i++) {
    for(int j = 0; j < matrix[i].length; j++) {
        System.out.println(matrix[i][j]);
    }
}
```

Column-Major Order

```
for(int i = 0; i < matrix[0].length; i++) {
    for(int j = 0; j < matrix.length; j++) {
        System.out.println(matrix[j][i]);
    }
}
```

Traversing With Enhanced For Loops

```
for(String[] rowOfStrings : twoDStringArray) {
    for(String s : rowOfStrings) {
        System.out.println(s);
    }
}
```