Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

### **MyClass.java**

Create a class called "MyClass" with a variable x:

public class MyClass {  
  int x = 5;  
}

## Create an Object

In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

### **Example**

Create an object called "myObj" and print the value of x:

public class MyClass {  
  int x = 5;  
  
  public static void main(String[] args) {  
    MyClass **myObj** = new MyClass();  
    System.out.println(myObj.x);  
  }  
}

## Multiple Objects

You can create multiple objects of one class:

### **Example**

Create two objects of MyClass:

public class MyClass {  
  int x = 5;  
  
  public static void main(String[] args) {  
    MyClass **myObj1** = new MyClass();  // Object 1  
    MyClass **myObj2** = new MyClass();  // Object 2  
    System.out.println(myObj1.x);  
    System.out.println(myObj2.x);  
  }  
}

## Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

* MyClass.java
* OtherClass.java

#### **MyClass.java**

public class MyClass {  
  int x = 5;  
}

#### **OtherClass.java**

class OtherClass {  
  public static void main(String[] args) {  
    MyClass **myObj** = new MyClass();  
    System.out.println(myObj.x);  
  }  
}

## Java Class Attributes

In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

### **Example**

Create a class called "MyClass" with two attributes: x and y:

public class MyClass {  
  int x = 5;  
  int y = 3;  
}

## Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (.):

The following example will create an object of the MyClass class, with the name myObj. We use the x attribute on the object to print its value:

### **Example**

Create an object called "myObj" and print the value of x:

public class MyClass {  
  int x = 5;  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass();  
    System.out.println(myObj.x);  
  }  
}

## Modify Attributes

You can also modify attribute values:

### **Example**

Set the value of x to 40:

public class MyClass {  
  int x;  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass();  
    myObj.x = 40;  
    System.out.println(myObj.x);  
  }  
}

### **Example**

Change the value of x to 25:

public class MyClass {  
  int x = 10;  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass();  
    myObj.x = 25; // x is now 25  
    System.out.println(myObj.x);   
  }  
}

### **Example**

public class MyClass {  
  **final** int x = 10;  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass();  
    myObj.x = 25; // will generate an error: cannot assign a value to a **final** variable  
    System.out.println(myObj.x);   
  }  
}

## Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

### **Example**

Change the value of x to 25 in myObj2, and leave x in myObj1 unchanged:

public class MyClass {  
  int x = 5;  
  
  public static void main(String[] args) {  
    MyClass myObj1 = new MyClass();  // Object 1  
    MyClass myObj2 = new MyClass();  // Object 2  
    myObj2.x = 25;  
    System.out.println(myObj1.x);  // Outputs 5  
    System.out.println(myObj2.x);  // Outputs 25  
  }  
}

## Multiple Attributes

You can specify as many attributes as you want:

### **Example**

public class Person {  
  String fname = "John";  
  String lname = "Doe";  
  int age = 24;  
  
  public static void main(String[] args) {  
    Person myObj = new Person();  
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);  
    System.out.println("Age: " + myObj.age);  
  }  
}

## Java Class Methods

You learned from the [Java Methods](https://www.w3schools.com/java/java_methods.asp) chapter that methods are declared within a class, and that they are used to perform certain actions:

### **Example**

Create a method named myMethod() in MyClass:

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

myMethod() prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

### **Example**

Inside main, call the myMethod() method:

public class MyClass {  
  static void myMethod() {  
    System.out.println("Hello World!");  
  }  
  
  public static void main(String[] args) {  
    **myMethod();**  
  }  
}  
  
// Outputs "Hello World!"

## Static or Public

You 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:

## Access Methods With an Object

### **Example**

Create a Car object named myCar. Call the fullThrottle() and speed()methods on the myCar object, and run the program:

// Create a Car class  
public class Car {  
   
  // Create a fullThrottle() method  
  public void fullThrottle() {  
    System.out.println("The car is going as fast as it can!");  
  }  
  
  // Create a speed() method and add a parameter  
  public void speed(int maxSpeed) {  
    System.out.println("Max speed is: " + maxSpeed);  
  }  
  
  // Inside main, call the methods on the myCar object  
  public static void main(String[] args) {  
    Car myCar = new Car();     // Create a myCar object  
    myCar.fullThrottle();      // Call the fullThrottle() method  
    myCar.speed(200);          // Call the speed() method  
  }  
}  
  
// The car is going as fast as it can!  
// Max speed is: 200

### **Example explained**

1) We created a custom Car class with the class keyword.

2) We created the fullThrottle() and speed() methods in the Car class.

3) The fullThrottle() method and the speed() method will print out some text, when they are called.

4) The speed() method accepts an int parameter called maxSpeed - we will use this in **8)**.

5) In order to use the Car class and its methods, we need to create an **object** of the Car Class.

6) Then, go to the main() method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).

7) By using the new keyword we created a Car object with the name myCar.

8) Then, we call the fullThrottle() and speed() methods on the myCarobject, and run the program using the name of the object (myCar), followed by a dot (.), followed by the name of the method (fullThrottle(); and speed(200);). Notice that we add an int parameter of **200** inside the speed() method.

Using Multiple Classes

Like we specified in the [Classes chapter](https://www.w3schools.com/java/java_classes.asp), it is a good practice to create an object of a class and access it in another class.

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

* Car.java
* OtherClass.java

#### **Car.java**

public class Car {  
  public void fullThrottle() {  
    System.out.println("The car is going as fast as it can!");  
  }  
  
  public void speed(int maxSpeed) {  
    System.out.println("Max speed is: " + maxSpeed);  
  }  
}  
**OtherClass.java**

class OtherClass {  
  public static void main(String[] args) {  
    Car myCar = new Car();     // Create a myCar object  
    myCar.fullThrottle();      // Call the fullThrottle() method  
    myCar.speed(200);          // Call the speed() method  
  }  
}

## 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:

### **Example**

Create a constructor:

// Create a MyClass class  
public class MyClass {  
  int x;  // Create a class attribute  
  
  // Create a **class constructor** for the MyClass class  
  public MyClass() {  
    x = 5;  // Set the initial value for the class attribute x  
  }  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass(); // Create an object of class MyClass (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.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

### **Example**

public class MyClass {  
  int x;  
  
  public MyClass(int y) {  
    x = y;  
  }  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass(5);  
    System.out.println(myObj.x);  
  }  
}  
  
// Outputs 5

### **Example**

public class Car {  
  int modelYear;  
  String modelName;  
  
  public Car(int year, String name) {  
    modelYear = year;  
    modelName = name;  
  }  
  
  public static void main(String[] args) {  
    Car myCar = new Car(1969, "Mustang");  
    System.out.println(myCar.modelYear + " " + myCar.modelName);  
  }  
}  
  
// Outputs 1969 Mustang

## Modifiers

By now, you are quite familiar with the public keyword that appears in almost all of our examples:

**public** class MyClass

The public keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

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

A screenshot of a social media post

Description automatically generated

A screenshot of a social media post

Description automatically generated

## Final

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

### **Example**

public class MyClass {  
  **final** int x = 10;  
  **final** double PI = 3.14;  
  
  public static void main(String[] args) {  
    MyClass myObj = new MyClass();  
    myObj.x = 50; // will generate an error: cannot assign a value to a **final** variable  
    myObj.PI = 25; // will generate an error: cannot assign a value to a **final** variable  
    System.out.println(myObj.x);   
  }  
}

## Static

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

### **Example**

An example to demonstrate the differences between static and public methods:

public class MyClass {  
  // Static method  
  static void myStaticMethod() {  
    System.out.println("Static methods can be called without creating objects");  
  }  
  
  // Public method  
  public void myPublicMethod() {  
    System.out.println("Public methods must be called by creating objects");  
  }  
  
  // Main method  
  public static void main(String[ ] args) {  
    myStaticMethod(); // Call the static method  
    // myPublicMethod(); This would output an error  
  
    MyClass myObj = new MyClass(); // Create an object of MyClass  
    myObj.myPublicMethod(); // Call the public method  
  }  
}

## 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: Person.java   
// abstract class  
abstract class Person {  
  public String fname = "John";  
  public int age = 24;  
  public **abstract** void study(); // abstract method   
}  
  
// Subclass (inherit from Person)  
class Student extends Person {  
  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: Person.java  
  
// Code from filename: MyClass.java  
class MyClass {  
  public static void main(String[] args) {  
    // create an object of the Student class (which inherits attributes and methods from Person)  
    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, you must:

* declare class variables/attributes as private (only accessible within the same class)
* provide public **setter** and **getter** methods to access and update the value of a private variable

## Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **getter** and **setter** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

### **Example**

public class Person {  
  private String name; // private = restricted access  
  
  // Getter  
  public String getName() {  
    return name;  
  }  
  
  // Setter  
  public void setName(String newName) {  
    this.name = newName;  
  }  
}

#### **Example explained**

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.

However, as the name variable is declared as private, we **cannot** access it from outside this class:

### **Example**

public class MyClass {  
  public static void main(String[] args) {  
    Person myObj = new Person();  
    myObj.name = "John";  // error  
    System.out.println(myObj.name); // error    
  }  
}

Instead, we use the getName() and setName() methods to acccess and update the variable:

### **Example**

public class MyClass {  
  public static void main(String[] args) {  
    Person myObj = new Person();  
    myObj.setName("John"); // Set the value of the name variable to "John"  
    System.out.println(myObj.getName());  
  }  
}  
  
// Outputs "John"

### **Example**

public class MyClass {  
  public static void main(String[] args) {  
    Person myObj = new Person();  
    myObj.name = "John";  // error  
    System.out.println(myObj.name); // error    
  }  
}

Why Encapsulation?

* Better control of class attributes and methods
* Class variables can be made **read-only** (if you omit the set method), or **write-only** (if you omit the get 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

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: <https://docs.oracle.com/javase/8/docs/api/>.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

### **Syntax**

import *package*.name.Class; // Import a single class   
import *package*.name.\*; // Import the whole package

## Import a Class

If you find a class you want to use, for example, the Scanner class, **which is used to get user input**, write the following code:

### **Example**

import java.util.Scanner;

In the example above, java.util is a package, while Scanner is a class of the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

### **Example**

Using the Scanner class to get user input:

import java.util.Scanner;  
  
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);   
  }  
}

## Import a Package

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (\*). The following example will import ALL the classes in the java.util package:

### **Example**

import java.util.\*;

## User-defined Packages

To create your own package, you need to understand that Java use a file system directory to store them. Just like folders on your computer:

### **Example**

└── root

└── mypack

└── MyPackageClass.java

To create a package, use the package keyword:

### **MyPackageClass.java**

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

Java Inheritance (Subclass and Superclass)

In Java, 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.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

### **Example**

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);   
  }  
}

## Java Polymorphism

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

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) 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.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

### **Example**

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");  
  }  
}

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

### **Example**

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 MyMainClass {  
  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();  
  }  
}

## Java Inner Classes

In Java, 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:

### **Example**

class OuterClass {  
  int x = 10;  
  
  class InnerClass {  
    int y = 5;  
  }  
}  
  
public class MyMainClass {  
  public static void main(String[] args) {  
    OuterClass myOuter = new OuterClass();  
    OuterClass.InnerClass myInner = myOuter.newInnerClass();  
    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:

### **Example**

class OuterClass {  
  int x = 10;  
  
  **private** class InnerClass {  
    int y = 5;  
  }  
}  
  
public class MyMainClass {  
  public static void main(String[] args) {  
    OuterClass myOuter = new OuterClass();  
    OuterClass.InnerClass myInner = myOuter.newInnerClass();  
    System.out.println(myInner.y + myOuter.x);  
  }  
}

## 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:

### **Example**

class OuterClass {  
  int x = 10;  
  
  static class InnerClass {  
    int y = 5;  
  }  
}  
public class MyMainClass {  
  public static void main(String[] args) {  
    OuterClass.InnerClass myInner = newOuterClass.InnerClass();  
    System.out.println(myInner.y);  
  }  
}

## Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

### **Example**

class OuterClass {  
  int x = 10;   
  
  class InnerClass {  
    public int myInnerMethod() {  
      return x;  
    }  
  }  
}  
  
public class MyMainClass {  
  public static void main(String args[]) {  
    OuterClass myOuter = new OuterClass();  
    OuterClass.InnerClass myInner = myOuter.newInnerClass();  
    System.out.println(myInner.myInnerMethod());  
  }  
}  
// Outputs 10

# **Java Abstraction**

Java Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

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:

An abstract class can have both abstract and regular methods:

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

### **Example**

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

## Java Interface

Another way to achieve [abstraction](https://www.w3schools.com/java/java_abstract.asp) in Java, is with interfaces.

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

### **Example**

// interface  
**interface** Animal {  
  public void animalSound(); // interface method (does not have a body)  
  public void run(); // interface method (does not have a body)  
}

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:

### **Example**

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

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

#### **Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

## Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

### **Example**

interface FirstInterface {  
  public void myMethod(); // interface method  
}  
  
interface SecondInterface {  
  public void myOtherMethod(); // interface method  
}  
  
// DemoClass "implements" FirstInterface and SecondInterface  
class DemoClass implements**FirstInterface, SecondInterface**{  
  public void myMethod() {  
    System.out.println("Some text..");  
  }  
  public void myOtherMethod() {  
    System.out.println("Some other text...");  
  }  
}  
  
class MyMainClass {  
  public static void main(String[] args) {  
    DemoClass myObj = new DemoClass();  
    myObj.myMethod();  
    myObj.myOtherMethod();  
  }  
}

**Java User Input**

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:

### **Example**

import java.util.Scanner;  // Import the Scanner class  
  
class MyClass {  
  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   
  }  
}

### **Example**

import java.util.Scanner;  
  
class MyClass {  
  public static void main(String[] args) {  
    Scanner myObj = new Scanner(System.in);  
  
    System.out.println("Enter name, age and salary");  
  
    // String input  
    String name = myObj.nextLine();  
  
    // Numerical input  
    int age = myObj.nextInt();  
    double salary = myObj.nextDouble();  
  
    // Output input by user  
    System.out.println("Name: " + name);   
    System.out.println("Age: " + age);   
    System.out.println("Salary: " + salary);   
  }  
}