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 are 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

You can have as many parameters as you want:

### 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

## 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:

### Example

// 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  
 }

}

# Java Encapsulation

## 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
* provide public **get** and **set** 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 **get** and **set** 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

}

}

If the variable was declared as public, we would expect the following output:

John

However, as we try to access a private variable, we get an error:

MyClass.java:4: error: name has private access in Person  
    myObj.name = "John";  
         ^  
MyClass.java:5: error: name has private access in Person  
    System.out.println(myObj.name);  
                  ^  
2 errors

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"

# Java Packages

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 uses 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!");

}

}

Save the file as **MyPackageClass.java**, and compile it:

C:\Users\*Your Name*>javac MyPackageClass.java

Then compile the package:

C:\Users\*Your Name*>javac -d . MyPackageClass.java

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

**Note:** The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users\*Your Name*>java mypack.MyPackageClass

The output will be:

This is my package!

# Java Inheritance

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 { // Super-Class [Parent]

protected String brand = "Ford"; // Vehicle attribute

public void honk() { // Vehicle method

System.out.println("Tuut, tuut!");

}

}

class Car extends **Vehicle** { // Sub-Class[Child] getting inherited

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");

}

}

Remember from the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) that we use the extends keyword to inherit from a class.

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();

}

}