studocu

Java Notes pdf

Object oriented programming (COSMATS University Sahiwal)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Java - What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

class

Fruit

objects

Apple

This document is available on studocu
Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Banana

Mango

Another example:

class

Car

objects

Volvo

Audi

Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

You will learn much more about classes and objects in the next chapter.

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.

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Create a Class

To create a class, use the keyword `class`:

Main.java

Create a class named "`Main`" with a variable x:

```
public class Main {
  int x = 5;
}
```

Remember from the Java Syntax chapter that a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

Create an Object

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

To create an object of `Main`, 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 Main {
  int x = 5;


  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
  }
}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of `Main`:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // 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:

- Main.java
- Second.java

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Main.java

```
public class Main {

  int x = 5;

}
```

Second.java

```
class Second {

  public static void main(String[] args) {

    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
5
```

ava 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:

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Example

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

```
public class Main {

  int x = 5;

  int y = 3;

}
```

Another term for class attributes is **fields**.

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 Main 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 Main {

  int x = 5;



  public static void main(String[] args) {

    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

Modify Attributes

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

You can also modify attribute values:

Example

Set the value of x to 40:

```
public class Main {
  int x;

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

Or override existing values:

Example

Change the value of x to 25:

```
public class Main {
  int x = 10;

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

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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

Example

```
public class Main {

  final int x = 10;


  public static void main(String[] args) {

    Main myObj = new Main();

    myObj.x = 25; // will generate an error: cannot assign a value to a
final variable

    System.out.println(myObj.x);

  }

}
```

The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

The `final` keyword is called a "modifier". You will learn more about these in the Java Modifiers Chapter.

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:

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Example

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

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // 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 Main {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

The next chapter will teach you how to create class methods and how to access them with objects.

Java Class Methods

You learned from the Java Methods chapter that methods are declared within a class, and that they are used to perform certain actions:

Example

Create a method named `myMethod()` in Main:

```
public class Main {
  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**;**

ExampleGet your own Java Server

Inside `main`, call `myMethod()`:

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

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

Try it Yourself »

Static vs. 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:

Example

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

```
public class Main {
  // Static method
  static void myStaticMethod() {
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    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 compile an error



    Main myObj = new Main(); // Create an object of Main

    myObj.myPublicMethod(); // Call the public method on the object

  }

}
```

Note: You will learn more about these keywords (called modifiers) in the Java Modifiers chapter.

Access Methods With an Object

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Example

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

```
// Create a Main class
public class Main {

  // 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) {
    Main myCar = new Main();   // 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!
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
// Max speed is: 200
```

Example explained

1) We created a custom `Main` class with the `class` keyword.

2) We created the `fullThrottle()` and `speed()` methods in the `Main` 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 `Main` class and its methods, we need to create an **object** of the `Main` 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 an object with the name `myCar`.

8) Then, we call the `fullThrottle()` and `speed()` methods on the `myCar` object, 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.

Remember that..

The dot (`.`) is used to access the object's attributes and methods.

To call a method in Java, write the method name followed by a set of parentheses **()**, followed by a semicolon (`;`).

A class must have a matching filename (`Main` and **Main.java**).

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Using Multiple Classes

Like we specified in the Classes chapter, 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:

- Main.java
- Second.java

Main.java

```
public class Main {

  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);

  }

}
```

Second.java

```
class Second {

  public static void main(String[] args) {

    Main myCar = new Main();      // Create a myCar object

    myCar.fullThrottle();       // Call the fullThrottle() method

    myCar.speed(200);           // Call the speed() method

  }

}
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
The car is going as fast as it can!
Max speed is: 200
```

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 Main class

public class Main {

  int x;  // Create a class attribute


  // Create a class constructor for the Main class

  public Main() {

    x = 5;  // Set the initial value for the class attribute x

  }


  public static void main(String[] args) {

    Main myObj = new Main(); // Create an object of class Main (This will
call the constructor)
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    System.out.println(myObj.x); // Print the value of x
  }
}

// Outputs 5
```

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like `void`).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

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 Main {
  int x;

  public Main(int y) {
    x = y;
  }
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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

// Outputs 5
```

You can have as many parameters as you want:

Example

```
public class Main {
  int modelYear;
  String modelName;

  public Main(int year, String name) {
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
// Outputs 1969 Mustang
```

Modifiers

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

```
public class Main
```

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

Access Modifiers

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a You will learn more about packages in the Packages chapter

This document is available on studocu
Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	**Description**
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifie learn more about packages in the Packages chapter
`protected`	The code is accessible in the same package and **subclasses**. You will learn more about subclass superclasses in the Inheritance chapter

Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	**Description**
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inherit

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

abstract	The class cannot be used to create objects (To access an abstract class, it must be inherited from a You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters

For **attributes and methods**, you can use the one of the following:

Modifier	**Description**
final	Attributes and methods cannot be overridden/modified
static	Attributes and methods belongs to the class, rather than an object
abstract	Can only be used in an abstract class, and can only be used on methods. The method does not h example **abstract void run();**. The body is provided by the subclass (inherited from). You will lea inheritance and abstraction in the Inheritance and Abstraction chapters
transient	Attributes and methods are skipped when serializing the object containing them
synchronized	Methods can only be accessed by one thread at a time
volatile	The value of an attribute is not cached thread-locally, and is always read from the "main memo

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Final

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

Example

```
public class Main {

  final int x = 10;

  final double PI = 3.14;


  public static void main(String[] args) {

    Main myObj = new Main();

    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 Main {
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
  // 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



    Main myObj = new Main(); // Create an object of Main

    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:

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Example

```
// Code from filename: Main.java

// abstract class
abstract class Main {

  public String fname = "John";

  public int age = 24;

  public abstract void study(); // abstract method

}


// Subclass (inherit from Main)

class Student extends Main {

  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: Main.java


// Code from filename: Second.java

class Second {

  public static void main(String[] args) {

    // create an object of the Student class (which inherits attributes
and methods from Main)

    Student myObj = new Student();
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    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`
- 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() {
```

This document is available on studocu
Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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 access and update the variable:

Example

```java
public class Main {

  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"
```

Why Encapsulation?

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

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

- 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

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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.

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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

```java
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

```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
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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 (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
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```java
  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);
  }
}
```

Did you notice the `protected` modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a `protected` access modifier. If it was set to `private`, the Car class would not be able to access it.

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The final Keyword

If you don't want other classes to inherit from a class, use the `final` keyword:

If you try to access a `final` class, Java will generate an error:

```
final class Vehicle {

  ...

}



class Car extends Vehicle {

  ...

}
```

The output will be something like this:

```
Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
                  ^
1 error)
```

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** 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.

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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

```java
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 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:

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

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 Main {
  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();
```

This document is available on studocu
Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    myPig.animalSound();

    myDog.animalSound();

  }

}
```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

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 Main {

  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    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 Main {
  public static void main(String[] args) {
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    OuterClass myOuter = new OuterClass();

    OuterClass.InnerClass myInner = myOuter.new InnerClass();

    System.out.println(myInner.y + myOuter.x);

  }

}
```

If you try to access a private inner class from an outside class, an error occurs:

```
Main.java:13: error: OuterClass.InnerClass has private access in
OuterClass
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
              ^
```

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 Main {

  public static void main(String[] args) {

    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

Note: just like `static` attributes and methods, a `static` inner class does not have access to members of the outer class.

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 Main {
  public static void main(String[] args) {
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
    OuterClass myOuter = new OuterClass();

    OuterClass.InnerClass myInner = myOuter.new InnerClass();

    System.out.println(myInner.myInnerMethod());

  }

}



// Outputs 10
```

Java Abstraction

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:

```
abstract class Animal {

  public abstract void animalSound();

  public void sleep() {

    System.out.println("Zzz");
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

Remember from the Inheritance chapter that we use the `extends` keyword to inherit from a class.

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");
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
  }
}


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

Why And When To Use Abstract Classes and Methods?

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

Note: Abstraction can also be achieved with Interfaces, which you will learn more about in the next chapter.

Java Interface

Interfaces

Another way to achieve abstraction 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)
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
  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");
  }
}
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
class Main {
  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).

ADVERTISEMENT

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

ExampleGet your own Java Server

```
interface FirstInterface {

  public void myMethod(); // interface method

}


interface SecondInterface {

  public void myOtherMethod(); // interface method

}


class DemoClass implements FirstInterface, SecondInterface {

  public void myMethod() {

    System.out.println("Some text..");

  }

  public void myOtherMethod() {

    System.out.println("Some other text...");

  }

}


class Main {

  public static void main(String[] args) {

    DemoClass myObj = new DemoClass();

    myObj.myMethod();

    myObj.myOtherMethod();

  }
```

This document is available on studocu

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)

```
}
```

Downloaded by Ashutosh Tiwari (tashutosh204@gmail.com)