Java for advanced:

Java OOP

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

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`:

Main.java

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

```
public class Main {

int x = 5;

}
```

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);

}

}
```

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

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);

}

}
```

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 "`Main`" with two attributes: `x` and `y`:

```
public class Main {

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 `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

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);
}
}
```

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);

}

}
```

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**;**

Example

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!"
```

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() {
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
}
}
```

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 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!
// 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)

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 Main {

int x;


public Main(int y) {

x = y;

}


public static void main(String[] args) {

Main myObj = new Main(5);

System.out.println(myObj.x);

}

}


// Outputs 5
```

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 sp modifier. You will learn more about packages in the Packages chapter

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 mo You will learn more about packages in the Packages chapter

`protected`	The code is accessible in the same package and **subclasses**. You will learn more about sub and 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 Inheritance chapter)
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited fr another class. 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 example **abstract void run();**. The body is provided by the subclass (inherited from). You w about 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 mem

Java File Handling

The `File` class from the `java.io` package, allows us to work with files.

To use the `File` class, create an object of the class, and specify the filename or directory name:

Example

```
import java.io.File;  // Import the File class


File myObj = new File("filename.txt"); // Specify the filename
```

The `File` class has many useful methods for creating and getting information about files. For example:

Method	Type	Description

`canRead()`	Boolean	Tests whether the file is readable or not
`canWrite()`	Boolean	Tests whether the file is writable or not
`createNewFile()`	Boolean	Creates an empty file
`delete()`	Boolean	Deletes a file
`exists()`	Boolean	Tests whether the file exists
`getName()`	String	Returns the name of the file
`getAbsolutePath()`	String	Returns the absolute pathname of the file
`length()`	Long	Returns the size of the file in bytes
`list()`	String[]	Returns an array of the files in the directory
`mkdir()`	Boolean	Creates a directory

Java Create and Write To Files

Create a File

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: `true` if the file was successfully created, and `false` if the file already exists. Note that the method is enclosed in a `try...catch` block. This is necessary because it throws an `IOException` if an error occurs (if the file cannot be created for some reason):

Example

```
import java.io.File;  // Import the File class

import java.io.IOException;  // Import the IOException class to handle
errors



public class CreateFile {

  public static void main(String[] args) {

    try {

      File myObj = new File("filename.txt");

      if (myObj.createNewFile()) {

        System.out.println("File created: " + myObj.getName());

      } else {

        System.out.println("File already exists.");

      }

    } catch (IOException e) {

      System.out.println("An error occurred.");

      e.printStackTrace();

    }

  }
```

```
}
```

The output will be:

```
File created: filename.txt
```

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

Example

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

Write To a File

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the `close()` method:

Example

```
import java.io.FileWriter;   // Import the FileWriter class

import java.io.IOException;  // Import the IOException class to handle
errors


public class WriteToFile {

  public static void main(String[] args) {

    try {

      FileWriter myWriter = new FileWriter("filename.txt");

      myWriter.write("Files in Java might be tricky, but it is fun
enough!");

      myWriter.close();
```

```
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Successfully wrote to the file.
```

Read a File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the `Scanner` class to read the contents of the text file we created in the previous chapter:

Example

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files


public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
```

```
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Files in Java might be tricky, but it is fun enough!
```

Delete a File

To delete a file in Java, use the `delete()` method:

Example

```
import java.io.File;  // Import the File class


public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
```

```
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

The output will be:

```
Deleted the file: filename.txt
```

Delete a Folder

You can also delete a folder. However, it must be empty:

Example

```
import java.io.File;


public class DeleteFolder {
  public static void main(String[] args) {
    File myObj = new File("C:\\Users\\MyName\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }
}
```

The output will be:

```
Deleted the folder: Test
```