

Study Guide For OOP in JAVA

Spencer Barnes

SID: 991728219

OOP in JAVA

Binsy Salin

Sheridan College

2/10/2024

Terms:	2
Setup Instructions	3
Types of High-level Programming Languages:.....	3
Key Takeaways:	4
Debugging:	4
JAVA Specifics:	5
Creating a Variable:	6
Starting a program:	6
Decision Structures:	7
Creating Objects:	9
Creating Classes:	9
Exception Handling:	11
Overloading:	12
Enumerators:	12
Putting it all together:.....	13

Terms:

- **Source Code** is the code that the programmer(you) write.
- **Machine Code** is a language that machines can directly understand, but contains only a combinations of 1's and 0's making it near impossible for humans to understand.
- **Bytecode** is a comprehensive and condensed set of instructions that are converted into machine code by a hybrid language interpreter on an end-users computer.
- **Syntax** described the rules you must follow given a certain programming language. It is the grammatical structure that the language follows.
- **Breakpoint** is a spot in your code where you would like the program to pause while running in debug mode.
- **JDK** – stands for Java Development Kit. It contains the Java compiler and the Java runtime environment.
- **Compiler** – Takes source code and converts it into either machine code or byte code, depending on the language.
 - Javac.exe is the Java compiler.
- **JRE** – stands for Java Runtime Environment. It is responsible for interpreting bytecode and running the JVM.
 - It contains the JVM and Java API.

- **JVM** – Stands for Java Virtual Machine. It is responsible for interpreting the bytecode and translating it into machine code.
- **Java API** – It is a set of instructions, or a library that the JVM can use to interpret bytecode.
- **Classes** are a set of blueprints that define how an object should be created.
- **Objects** are an instance of a class, created using the blueprints defined by that class.
- **Variables** are a stored value. They are allotted memory space to store a specific type of information. Could be a number(int or float), a boolean(bool) which are called **Primitive Variables** or a more complex custom datatype (Strings or an object variable).
 - Variables created inside a method are Local Variables, and will be destroyed when the method ends.
- **Casting** is converting a variable into a different type.
- **Constructor** is a method inside a class type that creates and assigns the variables needed inside the created object
- **Instance** is another term for a specific unique object created from a class. Other objects may be made from the same class but instances are distinct from one another.
- **Public Variables** are variables that are visible outside of the current scope
- **Private Variables** are only visible within the current scope
- Encapsulation is grouping data and methods into a class and controlling access with visibility modifiers like private, public, etc.
- **Abstraction** is concealing implementation details, defining a clear interface for interacting with objects, achieved through abstract classes and interfaces such as getters and setters
 - Getter: gets an instance variables value and returns it
 - Setter: sets an instance variables value with an argument value.
- **Pass by Value** – Java always passes VALUES when using arguments. Even when passing a reference to an object in java we are passing the VALUE of the address.

Setup Instructions:

- To install the Java Development Kit go to: <https://bell-sw.com/pages/downloads/> and download the FULL JDK.
- Once installed, go to Visual Studio Code extensions and search for: Extension Pack for Java'

Types of High-level Programming Languages:

- Fully interpreted
 - Is scripted, meaning it takes the source code and runs that on each machine.
 - Examples: JavaScript, Python, PHP
 - **Benefits:**
 - Easily run on any machine as it is translated at RUNTIME to run on a given machine
 - **Negatives:**
 - Slow to run as it much interpret each time it is run

- Fully Compiled
 - The end-user does not touch the source code, instead the source code is compiled once, converted into machine code and an executable file is created. This is what the end user receives when downloading the program.
 - Examples: C, C++
 - Benefits:
 - It is very fast, as no compiling or translation needs to occur at runtime.
 - Negatives:
 - Must be developed for each operating system individually, as the executable generated from the source code is platform specific.
- Hybrid
 - The source code is first compiled into Bytecode, this is what an end-user will download.
 - Examples: Java, C#
 - Benefits:
 - It runs quicker than a fully interpreted language
 - It runs equally on all machines that have the matching interpreter installed
 - Negatives:
 - The end-user MUST have the interpreter installed or the program will not run
 - Slightly slower than fully compiled languages as some conversion still takes place during runtime.

Key Takeaways:

All programming languages have vastly different syntax but share the same basic structure. All types of programming languages have their place and learning many of them will only benefit you.

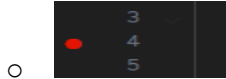
Some differences in syntax may include:

- Whether indentation is important
- Whether brackets are important [{}]
- Whether certain lines of code must be terminated with a semi colon [;]
- How verbose a language is.
 - the same operation of printing to the console in python versus in Java

Python	Java
<pre>src > python > app.py 1 print("Hello World!")</pre>	<pre>package helloworld; public class HelloWorld { public static void main(String[] args) { System.out.println("Hello World!!! My name is <<yo } }</pre>

Debugging:

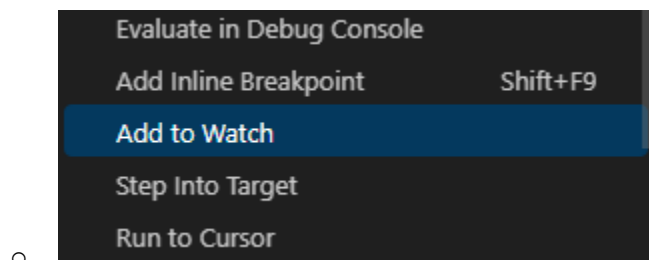
- To Debug code, first add a breakpoint by clicking along the margin of your code to the left of the line number, pictured below:



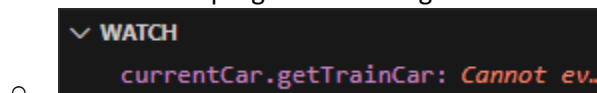
This will pause the program BEFORE it executes that line. Using the small bar that will appear at the top of your screen you can navigate through your code, these functions are shown below:



The debugger is essential in watching what exactly is happening in your code, it can display where the program is being run, which function it is inside, and can watch specific variables and their changes by using the “Add to watch” feature by right clicking on a variable while in Debug mode.



- Please note: The variable will not display a value if it has not been initialized at the point where you are paused at.
- **Tip:** use step over to watch how the values change each line, or step into to see which function the program is moving into.



JAVA Specifics:

A Good Rule of thumb for Java syntax is that if a line ends and the next character is NOT a “{”, terminate the line with a semi colon “;”. This tells the compiler that that argument is finished, and to move onto the next one.

HEAP, STACK, AND METHOD:

- Java stores programs and the methods it's running in three different locations these are called Heap, Stack and Method.
 - Method: The JVM loads the .class file into the method section. Essentially, it takes the bytecode and stores it here.
 - When running the JVM must always start with the **main** method.

- Stack: The stack is the area where the JVM stores the data for methods that are currently active. It further separates them into Stack Frames, which are slices that contain only a single method, and are “stacked” with the first accessed method at the bottom and the currently active method on the top. As a method resolves, it will then POP that Stack Frame off the Stack and move to the next Stack Frame.
 - As this happens ANY variables created in the stack frame are destroyed, unless they are passed back to the previous function, or are stored in the heap.
- Heap: This is where the JVM stores things that need to be kept around for longer than a single method. Objects are an example of this.

Creating a Variable:


When creating a variable in Java the first step is to decide what type of value you need to store. If you need a number with no decimals, perhaps an “int” would suffice, or maybe you need to store a true or false value, then you would choose “bool”. Then you need to decide a name for the variable. Try to choose a name that suits the purpose of the variable and format it with no spaces, using either the camelCase or underlined_style. For a variable the first letter should be lowercase.

○

```
int demo_of_an_int = 5;
```

Sometimes you can take a less specific variable type and **CAST** it to a different type, for example you can take an int and make it into a double(number with a decimal). This is done by assigning the int to a double:

○

```
29 | int i=200;  
30 |  double d=i;
```

This is called **Implicit Casting**, and happens automatically, as there is no loss of data during the conversion, however casting the other way will not work. In order to cast a more complex variable into a simpler one, you need to **Explicitly Cast** that variable like so:

○

```
double myDouble = 9.78d;  
int myInt = (int) myDouble;
```

Note that in both cases, we cannot assign the variable to itself in a different type, we must create a new variable of the converted type.

Starting a program:

Usually a java project will automatically create a .java file with the package name and the class set up, like so:

```
package week5Lab;

You, 3 days ago | 1 author (You)
public class Start {
```

From there, you must declare the main function like so:

```
public class Start {
    |
    Run | Debug
    public static void main(String[] args){
```

Notice that the main function is INSIDE the {} of the Class. From there you can declare your variables, call methods, create object, and innovate as you please.

Decision Structures:

So far we have been working sequentially. But what if we want to have multiple outcomes for our code given different circumstances?. Introducing Decision Structures.

IF, ELSE IF, ELSE:

- The first and most easy to understand decision structure is the if, else if, else statement shown below:

```
if (20 > 18) {
    System.out.println("20 is greater than 18");
}
else if (20 == 20){
    System.out.println(x:"20 is still 20!");
}
else{
    System.out.println(x:"Math has been wrong all along!");
}
```

This structure allows you to set up multiple conditions that will sequentially check whether the conditions evaluate to true. These **always resolve to a true or false value** in the end and the system will execute **only the first true statement** in an if block(if you provide another IF it will create a new block separate from the first one. In the above example, the first condition will always resolve to true, and **the other two will never be called**.

- You can also use multiple expressions in a single if statement by taking advantage of the AND(&&) and OR(||) operators allowing you to chain together complex decisions into a single line.

```
int myNumber = 15;  
if (myNumber > 10 && myNumber < 20){  
    System.out.println(x:"the number is between 10 and 20");  
}
```

This also takes use of the greater than(>) or less than(<) operators, but there are more such as but not limited to:

- Equal to - ==
- Not equal to - !=
- Modulus - %
- Arithmetic – minus(-), plus (+), divide (/), multiply(*)

Loops:

- There are three types of loops:
 - While loops
 - Using an while statement with what is called a **Sentinel Value** and modifying that value inside the loop allows you to perform a task a specific number of times.
 - ```
int counter = 0;
while (counter < 5) {
 System.out.print(counter + ", ");
 counter++;
}
```
  - Do While loops
    - Is a loop that primes itself. In other words, regardless of the condition being met, it will perform the contents of the loop once and then check the condition on whether to restart the loop.
    - ```
int num = 0;  
{  
    System.out.println(num);  
    num++;  
}  
while (num < 4);
```
 - For loops
 - These are a bit more tricky, First you declare a sentinel value inside the loop declaration, you provide the condition for the loop, and then you provide a modifier for your sentinel value. Its easier to look at it and understand.


```
for (int Sentinel = 0; i < 5; Sentinel++) {  
    System.out.println(Sentinel);  
}
```

This is performing nearly the exact same thing as the above loops, but in a shorter more condensed form.

Creating Objects:

Before we can start creating our own classes, we must first learn how to create an object instance of a custom class, and luckily its similar to how we declare a variable, with some key differences.

- `Student s1 = new Student();`

This is an object declaration. Note that like declaring an int, we start with the type of variable we are creating. We then name the object like we would a variable, but after the assignment operator(=) we must use the keyword "new" which tells the compiler that we want to create a new object. After that we call the constructor for that type in this case Student().

- Note that Student() looks similar to main(String[] args) that is because it is a method just like main, but it is defined inside of the Student class.

It is also important to note that Objects and the variables inside objects are stored inside the HEAP. And variables created to hold an object is actually holding a reference that points toward the object stored in heap.

Creating Classes:

Inside a class there can be entire programs. There may be variables which are shared between every Instance, Instance specific variables, or even methods.

In general, we want to hide the inner workings of our classes away from the public, and thus, unless necessary, it is best practice to hide most of our class variables by using the private keyword when declaring them. This necessitates the need for getter and setters. These are functions that GET and SET our variables, without allowing direct access to those variables.

```
package week5Lab;

You, 3 days ago | 1 author (You)
public class TrainCar {
    private TrainCarType type;
    private TrainCar nextTrainCar = null;

    TrainCar(){}
    week5Lab.TrainCar.TrainCar(TrainCarType type)
    TrainCar(TrainCarType type){
        this.type = type;
    }
    public void setTrainCar(TrainCar nextTrainCar){
        this.nextTrainCar = nextTrainCar;
    }
    public TrainCar getTrainCar(){
        return nextTrainCar;
    }
    public TrainCarType getType(){
        return type;
    }
}
```

- This is an example of a class declaration, note the private variables and the getType() and setTrainCar() methods, which get or set the values of each variable. Some methods may even be private as well, only being called by other class methods.

This keyword:

Take note in the above picture of the use of the this keyword. The this keyword essentially tells Java that the variable directly after it is an instance variable and may not be the local variable that is defined within the current method. Doing this allows you to declare a variable that is the same name as class variable in the set method without having to worry about how you will assign that variable to the class defined variable of the same name.

```
public void setTrainCar(TrainCar nextTrainCar){
    this.nextTrainCar = nextTrainCar;
}
```

-

Always remember. When you create a class, it will not perform its function unless called within the main function or through another function called in the main function. But only one main function is required in each program.

Static Keyword:

The static keyword is a keyword used in method or variable declarations that say that this thing is not modified. It has one value. If a variable is declared static in a class, all objects of that class will share it. Also, static variables and methods can be accessed without creating an object of that class type. A method that modifies instance variables should not be static.

In general: if a method can be static it should be, but only declare a variable static if it needs to be.

Exception Handling:

Throwing Exceptions:

While writing a program, it is a good idea to provide opportunities for your program to exit safely instead of crashing upon encountering an error. We do this by declaring exceptions. These allow the program to report an exception and even continue after detecting the error and attempting a fix.

When writing a method, if the method is going to throw an exception, you must declare that in the method declaration

```
public static void main(String[] args) throws Exception
```

Do this by writing “throws Exception” between the arguments and the method body.

```
int demo_of_an_int = 5;
if (demo_of_an_int == 5)
{
    Exception err = new Exception(message:"wrong number!");
    throw err;
}
```

If a method returns an error you can choose how to proceed from there. You can also choose to handle an error in any way that fits the situation but a common example could be retrying to get an input from a user.

Handling exceptions by throwing them leaves dealing with those exceptions to the caller, instead of dealing with them inside the method itself.

Try/Catch:

Similar to the if/else if/else structure, the try/ catch/ finally structure attempts to run the try section, and then upon encountering an exception will catch specific exceptions in each catch section, and then upon reaching finally will run the code contained there regardless of anything.

```
try {  
    int result = 10/0;  
    System.out.println("Result: " + result);  
} catch (ArithmeticException e) {  
    System.out.println(x:"Error: Division by zero");  
} finally {  
    System.out.println(x:"Inside finally block");  
}
```

Overloading:

Overloading a method is when you define a method multiple times with different arguments. Perhaps a customer only provides their name and their email address but not their phone number, we can still make an account using that information so we provide an alternative constructor for the customer object. An example of overloads:

```
//let's overload study to study a particular subject  
public void study(String sub)  
{  
    System.out.println(name + " is studying "+ sub);  
}  
//let's further overload study so we can also choose how many hours to study  
public void study(String sub, int hours)  
{  
    System.out.println(name + " is studying " + sub + " for " + hours+ " hours");  
}  
public void study(String sub1,String sub2)  
{  
    System.out.println(name + " is studying " + sub1 + " and " + sub2);  
}  
}
```

Here we define three different study() methods all with different variables passed in as arguments. This allows us flexibility in how we allow people to use our programs. When differentiating notice that the input arguments never have the same TYPES. having two study() methods that both take two String values would cause an error because when passing in two String values the program would not know which of the two definitions to use.

Enumerators:

Enumerators allow us to set a list of constant values in a list, and create values that point to specific items in said list. We can even give the list elements variables. Enums must contain a constructor for any variables that the list elements are given. And example of an Enum is shown below:

```
package Enums;

public enum AnimalTypes{
    ELEPHANT(size:"large"),MONKEY(size:"small"),BEAR(size:"medium"),LION(size:"medium"),TIGER(size:"medium");

    String size;

    AnimalTypes(String size){
        this.size = size;
    }
}
```

○ Notice that the constructor is declared with the same name as the Enum is and has no additional modifiers before it.

Putting it all together:

Until now classes have only been composed of primitive variables, but what if you filled a class with objects of other classes?

What this results in is an object that holds reference values to other objects or enums. An example is shown below:

```
package week5Lab;

public enum TrainCarType {
    LIQUIDSTORAGE(weight:2000),DRYSTORAGE(weight:1500),FROZENSTORAGE(weight:2500);

    private int weight;

    TrainCarType(int weight){
        this.weight = weight;
    }

    public int getWeight(){
        return weight;
    }
}
```

○ First we declare an Enum called TrainCarType and give it values and even make a getter for it.

```
package week5Lab;

You, 3 days ago | 1 author (You)
public class TrainCar {
    private TrainCarType type;
    private TrainCar nextTrainCar = null;

    TrainCar(){}

    TrainCar(TrainCarType type){
        this.type = type;
    }
    public void setTrainCar(TrainCar nextTrainCar){
        this.nextTrainCar = nextTrainCar;
    }
    public TrainCar getTrainCar(){
        return nextTrainCar;
    }
    public TrainCarType getType(){
        return type;
    }
}
```

- Then we create a class called TrainCar, but notice that both variables are holding objects of a class or an Enum.