Java Foundations Notes

Chapter 1

CSC110

# Java Syntax & Semantics

A Java **program** is made up of one or more classes. A **class** is a template for the features of **objects** which contains members (methods & attributes). A **method** is a procedure that is tied to an object. It contains local variables and statements. Attributes (also called fields or properties) are the data members of a class. A Java **application** contains a method called main().

The **syntax** rules of a language define the legal ways to arrange the identifiers and symbols (**tokens**) of a language to make a valid program. The **semantics** of a program statement define what that means. An incorrect program can still be syntactically correct! A program does what you tell it to do, not what you want it to do. The distinction between syntax and semantics can sometimes be hard to grasp at first. Roughly, think of syntax as the thing your compiler checks, while semantics are what you check when you test your code.

### Identifiers

An **identifier** is the name given to the entities of a program, such as variables, classes, methods, packages, labels, constants, etc. In Java, identifiers are made up of letters, digits, and the symbols \_ and $. Identifiers are case sensitive and cannot begin with a digit. That means that Total, total, and TOTAL are three distinct identifiers. (they are also an indication of a programmer who is not using a good coding style!).

Java programmers typically follow these naming conventions:

* Capitalize class names
* ALL CAPS for constants
* lowercase names for packages
* camelCaseNames for method and variable names

## Coding Style

Whatever your particular coding style, you should always strive for simplicity. Using coding tricks and clever code can be fun, until you have to decipher it later. Any time you use an advanced feature of your language, always ask yourself if you could be doing it in a simpler way. Code has a way of changing over time and simple code is *much* easier to change that clever code.

The most important thing about your coding style is that you are consistent. Select an indention style and naming convention and **stick to it** throughout your code. Java ignores extra whitespace, so use it to improve readability.

I personally could not possibly care less about your coding style. In the code examples for this class I try to use 4-spaces to indent (not a tab), though I use a lot of different editors over time while editing the code so that may not always be consistent.

Use **comments** for inline documentation. They do not affect how the program works, but they can help others understand how your code works. However, don’t go overboard with your comments - make sure they are all helpful to another programmer without being too verbose. Keep in mind that variable and method names can also be used to write clear code. You should try to write code that does not need to be documented. Comments should be used only to provide the big picture.

A few other things to keep in mind that we will expound on later:

* Classes/methods should encapsulate/do one thing
* Provide tests to demonstrate how your code works
* Bad programmers write complex code, good programmers write simple code.

*Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*

*Code for readability.*

- John F. Woods, comp.lang.c++

#### Bad code style example

Notice how in the example below, none of the variable or method names describe what the thing is. And, even worse, the comments just describe the syntax, not the purpose of the code. I would argue that in this example the only reason the comments seem to be there are to make the code *even harder to read*!

##### int num = 0; // declare int

##### int num2 = 0; // declare int

##### // Call some function

##### num = someFunction();

##### // If num is more than 100 call doSomething

##### if (num > 100) { doSomething();}

##### // else call doSomethingElse

##### else {doSomethingElse();

##### num2 ++; // increment ++

##### }

##### // for 0 to count call prepareSomeStuff

##### for (ii=0;ii<num;ii++) {

##### prepareSomeStuff();

##### }

**Better code style example**

The example below is exactly the same as the code above other than the variable and method names have been changed, the comments updated, and better use was made of white space. Notice how the code is much easier to understand *even though there are fewer comments*.

##### int errorCount = 0; // Number of errors

##### int successfulCount = 0; // Number of jobs with acceptable error rate

##### 

##### errorCount = getErrorCount();

##### 

##### // Acceptable error count is less than 100

##### if (errorCount > 100) (

##### reportExcessiveErrors();

##### }

##### else {

##### releaseForProcessing();

##### successfulCount ++;

##### }

##### 

##### // Clear errors from system to prepare for next run

##### for (ii=0;ii<errorCount;ii++) {

##### resetError();

##### }

# Programming Notes

**Errors**

There are different types of errors you can run across. **Compile-time errors** occur when you try to compile your program. These are typically syntax errors or type errors. **Run-time errors** occur when your code runs. They typically lead to exceptions and your program will crash if the exception is not caught. **Logical errors** are bugs, not in your code itself, but in the design of your program. These are the worst kind of errors since they can only be detected by thorough testing. Compile-time errors are the best kind of errors since they are easiest to spot, if not always to fix.

**Programming Pitfalls**

As you write your code, try to avoid the following types of (very common) errors that programmers make:

* Random Walk Programming is making small, random changes until your code compiles. This is a bad habit! When you find an error you should always try to figure out why the error occurred and then fix it in a way that ensures it is really fixed. Keep in mind that just because your code compiles that doesn’t mean it will work.
* Cargo Cult Programming / Magical Incantations is copying and pasting code you don’t understand. If you don’t understand code you should *never* include it - it could be dangerous.
* Big Ball of Mud is “A haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle.” Big balls of mud usually result when you start programming before fully understanding the requirements without a good design and frequently incorporate the 2 problems above.

# Program Development Process

## Steps in the Development Process

The programming development process consists of multiple steps. In practice, there is a lot of overlap between the steps and there may be cycles where consecutive steps are repeated multiple times. The important part here is not that you always rigidly follow the steps below, but that, at some point while your code is under development, you visit each step. Notice that actual coding is a small part of the overall process.

* The **requirements** are an informal description of the program’s needed functionality from the user’s point of view.
* A **specification** is a formal description of the program's requirements from the programmer’s point of view.
* The **design** process entails translating the specification into procedures.
* **Implementation** involves actually instantiating the design (programming)
* **Verification** ensures that the implementation meets the requirements.
* **Validation** ensures that the implementation meets the specification.

## Program Testing

**Program testing** is executing a program by applying test cases in an attempt to find programming errors in a given environment. **Test cases** a set of inputs to a program and the expected output given those inputs. Test cases are created based on pre/post conditions. Keep in mind that exhaustive testing (testing all possible inputs) is rarely feasible. Your test cases should check for edge cases - situations that are especially difficult for your program to handle. These may be things such as negative numbers entered as dates, strings entered when integers are expected, or adding elements to the start or end of a data structure.

Programmers often will create a **driver class** to test a class. A driver class is a class that usually has just a main method. The main method of a driver class instantiates one or more instances of the class you are testing and exercises the methods to verify the class works as expected. Output or exception handling can be used to detect errors. Note that there is nothing magical about a driver class or the name driver. Driver classes are not required, but they can be a helpful tool on your programming toolbelt.

**Debugging** is the process of finding the location and root cause of program errors.

### Types of Testing

**Functional Testing** (Black-box testing) covers some subset of the requirements of the program. The idea is to ensure the program does what it is supposed to do - the output is correct for a given input. The test cases should be implementation independent, so if the implementation changes the test cases should still be applicable.

**Structural Testing** (White-box testing) covers the structure of the program with an aim of discovering if there are any bugs in the program code. With structural testing, test cases are aware of the implementation so if there are changes to the implementation the test cases have to change.

## Types of Program Maintenance

With most programs, especially in industry, you will need to perform maintenance to ensure that your program continues to meet the requirements. There are many different types of maintenance. **Corrective maintenance** is fixing bugs and other problems as they crop up. **Adaptive maintenance** is required when the execution environment changes (OS upgrades, new technology). **Perfective maintenance** involves implementing new features or enhancements based on user requests. Finally, **preventive maintenance** involves proactively making bug fixes and making enhancements to improve maintainability. You should design your programs with these in mind to make your future self happy.

### Proof of Program Correctness

There are two factors in proving a program is correct

* 1. **Partial Correctness**
     + - * Does the program satisfy the preconditions?
         * Does the program satisfy the postconditions?
  2. **Termination**
     + - * Does the program end?

# A program is considered to be correct when it meets the conditions of both partial correctness and termination.

# Overview of Object-Oriented Programming

Java is an **Object-Oriented Programming (OOP)** language. (OK, that isn’t 100% true, but it is close enough for our purposes.) We will cover OOP in more detail later in the semester, but I want to give you an overview now so you can see where we are going. I don’t expect you to fully understand this material now, but this is what we are building towards in this course.

In Object-Oriented Programming, the fundamental entity is the “**object**”. An object has some information (**state**) & some operations (**behaviors**) and usually represents some real-world entity such as:

* A particular student in a class
* A window in a GUI
* A character in a game

Objects should handle their own processing and data management.

Object-Oriented languages vary widely, but typically the contain the following features, often called “The Four Pillars of Object-Oriented Programming”:

* **Abstraction** - shifting the focus on what an object does instead of the details of its implementation
* **Encapsulation** - separating the interface from the implementation
* **Inheritance** - subclasses can include attributes & methods from a superclass
* **Polymorphism** - methods get called based on the type of the object no matter what the type of the reference is (late-binding)

The explanations above are by necessity fairly hand-wavy for now. We will discuss each of these in more detail later.

In Object-Oriented Programming there are usually multiple ways to represent a problem and there is often no “one right answer”. In fact, in many cases OOP itself is not always the answer (though for purposes of this class it will nearly always be). In practice, there are usually multiple ways to solve a problem, with a few of them being good choices. There are also lots of ways to poorly solve a problem, which is why experience is your best friend when it comes to programming. The more you practice the better you will get.