Programming Languages (Outsource: 2-2)

A computer **program** is a set of instructions that you write to tell a computer what to do. There are many languages used to write computer programs, each language with a specific purpose.

"So this software... Does it tell you to do things?"

There are two kinds of programming languages: **high-level languages** and **low-level languages**. Computers actually only understand one language – machine language which is the most basic circuitry-level language. Machine language is very difficult to program in because it consists of nothing but binary numbers – 0's and 1's.

To simplify the task of programming a number of **high-level programming languages** have been developed. A high-level programming language allows you to use a vocabulary of English like words such as "print" or "read" instead of the sequences of binary numbers that actually perform these tasks. High level languages also allow you to assign names to memory locations to store information rather than having to remember the exact memory locations (addresses). High-level programming languages include BASIC, Pascal, Lisp, FORTRAN, C++ and Java. Since a computer isn't capable of understand high-level languages a **compiler** or **interpreter** is used to translate the high-level language into machine language.

Each high-level language has its own **syntax**, or rules of the language. For example, depending on the specific high-level language, you might use the verb "print" or "write" to produce output. All languages have a specific, limited vocabulary and a specific set of rules for using that vocabulary.

In addition to learning the correct syntax for a particular language, a programmer also must understand computer programming logic. The **logic** behind any program involves executing the various statements and procedures in the correct order to produce the desired results.

The Java programming language was developed by Sun Microsystems as an object-oriented language that is used both for general-purpose business programs and interactive World Wide Web-based Internet programs. Some of the advantages that have made the Java programming language so popular in recent years are its security features, and the fact that it is

architectuarally neutral, which means that you can use the Java programming language to write a program that will run on any platform. A program written in the Java programming language is compiled into Java Virtual Machine code, called **bytecode**. The compiled bytecode is subsequently interpreted on the machine where the program is executed. Any compiled program will run on any machine that has a Java programming language interpreter.

Another advantage of the Java programming language is that it is simpler to use than many other object-oriented languages. The Java programming language is modeled after the C++ programming language. Although neither language qualifies as "simple" to read or understand on first exposure, the Java programming language eliminates some of the most difficult features to understand in C++.

Object-oriented Programming (Outsource: 9-2)

Object-oriented programming, OOP for short, is a computer programming model where the programmer envisions program components as objects that are similar to concrete objects in the real world.

The idea behind object-oriented programming is that a computer program may be seen as comprising a collection of individual units, or objects, that act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine or actor with a distinct role or responsibility. Procedural to OOP may help understanding the concept using code.

Object-oriented programming is claimed to promote greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. Furthermore, proponents of OOP claim that OOP is easier to learn for those new to computer programming than previous approaches, and that the OOP approach is often simpler to develop and to maintain, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than other programming methods. Critics dispute this, at least for some domains (industries).

Classes (Outsource: 9-1 – 9-11)

Classes are a fundamental part of the Java programming language. No methods can occur outside of a class and no statements may occur outside of a method. Be aware that every program must begin with a class definition and the name of the class containing the **main** method should be the same as the name of the program file. This is very important. If the name of the class does not match the name of the file, your program will not execute properly.

Creating a Class (Outsource: 10-8)

When you create a class, first you must assign a name to the class, and then you must determine what data and methods will be part of the class. To begin, you create a class header with three parts:

- An optional access modifier
- The keyword class
- Any legal identifier you choose for the name of your class

You can use the following class access modifiers when defining a class: **public**, **final**, or **abstract**. If you do not specify an access modifier, access becomes **friendly**.

Public classes are accessible by all objects, which means that public classes can be **extended**, or used as a basis for any other class. The most liberal form of access is public. You will use the public access modifier for most of your classes.

After creating the class header you write the body of the class between a set of curly brackets. The body of the class is made up of any number of variables and methods.

The allowable field modifiers for instance variables are **private**, **public**, **friendly**, **protected**, **private protected**, **static**, and **final**. *Most class fields are private*. Private access is sometimes called **information hiding**. A class's private data can be changed or manipulated only by a class's own methods, and not by methods that belong to other classes. In contrast, most class methods are not usually private. The resulting private data/public method arrangement provides a means for you to control outside access to your data. Only a class's non-private methods can be used to access a class's private data.

The following requirements must be met when creating the name for a class:

- A class name must begin with a letter of the alphabet, an underscore, or a dollar sign.
- A class name can contain only letters, digits, underscores, or dollar signs.
- A class name cannot be a Java programming language reserved keyword, such as *public* or *class*.
- A class name cannot be one of the following values: **true**, **false**, or **null**.

Methods (Outsource: 10-11 – 10-13)

A **method** is a series of statements that carry out some task. Any class can contain an unlimited number of methods. Within a class, the simplest methods you can invoke do not require any arguments or return any values. For example, consider the simple Hello World program:

```
public class HelloWorld
{
    public static void main( String args[ ] ) {
        HelloWorld world = new HelloWorld();
        world.output();
    }

    public void output() {
        System.out.println("Hello World");
    }
}
```

JAVA IS CASE SENSITIVE!	You must be very careful about distinguishing between upper case and lower case letters.

There are two reasons to use methods in a program. First, it makes it easy to follow the flow of a program. In other words, your programs become more readable and easier to understand. More importantly, methods are *reusable*.

Every method must include the following:

- A declaration (or header or definition)
- An opening curly bracket
- A body
- A closing curly bracket

The method declaration contains the following:

- Optional access modifiers
- The return type for the method
- The method name
- An opening parenthesis
- An optional list of method arguments (you separate the arguments with commas if there is more than one)
- A closing parenthesis

The access modifiers for a method can be any of the following modifiers: **public**, **private**, **protected**, **private protected**, or **static**. Usually methods are given **public** access. Public means that the method can be accessed from any class. Declaring a method to be **static** means that it is a class method rather than an object method. Static methods can only call other static methods and operate on static variables.

The statement

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

defines a **method** called **main**. A method contains a collection of programming instructions that describe how to carry out a particular task. Every Java application must have a main method. Most Java programs contain other methods besides main.

The parameter **String args[]** is a required part of the main method. It contains *command line arguments*. The keyword **static** indicates that the main method does not operate on an **object**. This will be explored later when we begin discussing classes in earnest. **Void** means the method returns no values and **public** means it can be called from outside the class. For now, simply be aware that all Java applications will begin in a similar fashion.

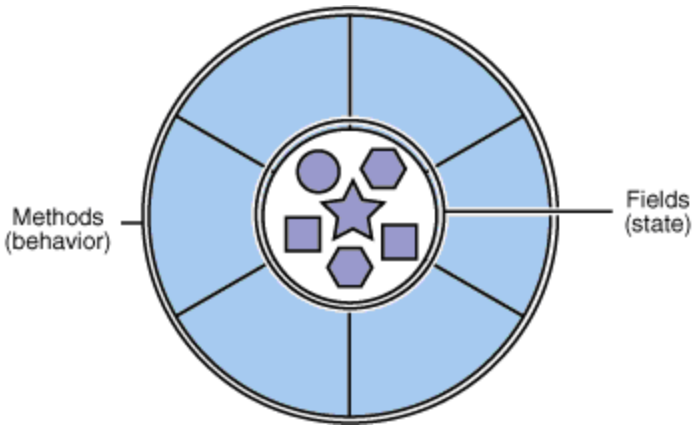
Quentin Tarantino's "Learn Java in a Minute"

What is an Object? (Outsource: 10-1 – 10-2)

Objects are the key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have states (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Sprites also have states (locations (x and y) and Images) and behavior (moveLeft, moveRight, etc). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

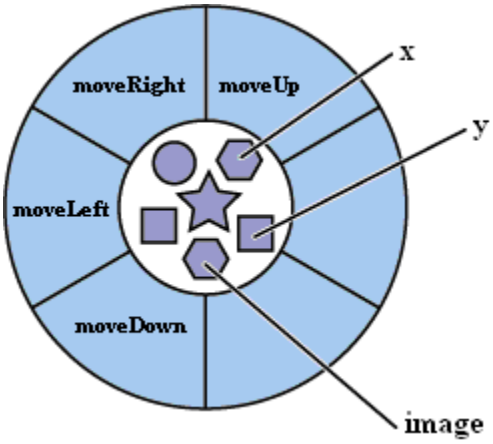
Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data **encapsulation** — a fundamental principle of object-oriented programming.

Consider a sprite, for example:



A bicycle modeled as a software object.

By attributing state (`x`, `y`, and `image`) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the sprite is not allowed to go off the screen, the move method's could test for current location reject any calls that decrease the x or y values to less than 0 or increase them to greater than the screen width/height.

Bundling code into individual software objects provides a number of benefits, including:

1. *Modularity*: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. *Information-hiding*: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. *Code re-use*: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. *Pluggability and debugging ease*: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Instantiating Objects (Outsource: 10-7 – 10-8)

Declaring a class does not create any actual objects. A class is just an abstract description of what an object will be like if any objects are ever actually instantiated.
A two-step process creates an object that is an instance of a class. First, you supply a type and an identifier, just as when you declare any variable, and then you allocate computer memory for that object. To allocate the needed memory, you must use the **new operator**. The new operator allocates a new, unused portion of computer memory and returns a memory address. You do not need to be concerned with what the actual memory address is – but you do need to be aware that every object name is also a reference – that is, a computer memory location. Next, you make a call to the **constructor method** of the class you are instantiating. A constructor method is a method that is used to construct an object, or initialize an object. If you don't create a constructor when you define a class then Java automatically creates one for you. The name of the constructor method is always the same as the class name. Like any other method, constructors can be overloaded by declaring multiple constructors with different arguments.

After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call. For example, the String class has a method called `charAt(int index)` which takes an integer argument (the position of the desired character). To use the method you must declare an object of the String class and then call the method:

```
String str = new String("Hello World!");
char ch = str.charAt(4);
```

Top-Down Design

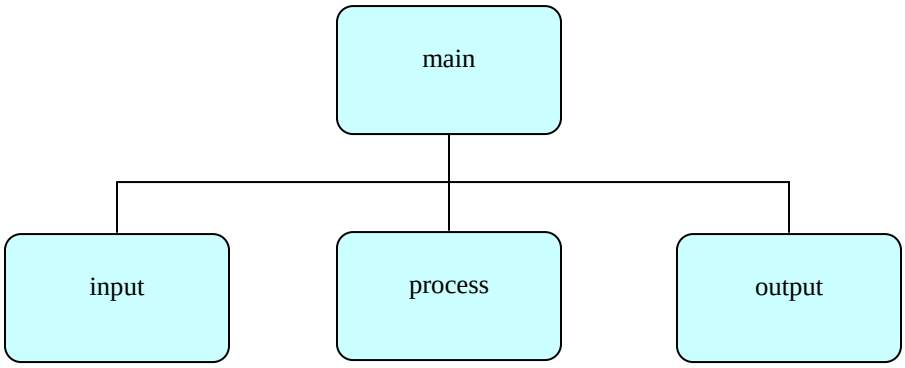
There are many approaches to the construction of large computer programs. One common method is the **top-down design**. In the top-down approach, the programmer first conceives of the major tasks to be accomplished (the "top") and writes the main() method with the major tasks as method calls. Each major task is refined into smaller subtasks (methods). These subtasks may, in turn, be further refined. The entire process continues until the resulting subtasks are single, specific tasks that are easily translated into Java code.

Top-down design has many advantages. It helps the programmer (and whoever may need to read the program) to keep things organized. It allows difficult, lower-level coding to be done after the

major idea has been clearly specified. It makes it possible to get a program running quickly, and then to progressively add features. Finally, it allows the program to be broken up into specific tasks hat can be worked on and tested separately from the whole, and possibly even by different programmers.

Structure Charts

A structure chart graphically represents the top-down refinement process. Each level represents further refinement of the problem into smaller sub problems. Level 0 contains the general statement of the problem (the **main()** method), Level 1 contains the first level of refinement, and so forth. There may be as many levels of refinement as needed to break a problem into manageable subtasks. The following example shows the minimum methods needed for a typical interactive program. It contains a **main()** method that calls three other methods: **input()**, **process()**, and **output()**.



Console Output in Java

Console output in a Java application is accomplished by calling the **System.out.print(*argument*)** and **System.out.println(*argument*)** statements. Both statements output a string of characters to the screen. The difference between print and println is that the println statement will cause the cursor to advance to the next line while print leaves the cursor where it is. Calling **System.out.println()** without passing it a string argument will simply send the cursor to the next line. Attempting to call **System.out.print()** without a string argument will cause a **syntax** error.

System.out.println

In Java, you can print a **String** to the screen.

```
System.out.println( "Hello, World!" ) ;
```

Or an **int**.

```
System.out.println( 42 ) ;
```

Or a **double**.

```
System.out.println( 3.14159265358979 ) ;
```

In fact, Java allows you to print almost any type. Whether it does something sensible depends on the type. It usually handles all of the primitive types well, but has some problems with object types.

As you can tell, to print something you need:

```
System.out.println( STUFF ) ;
```

That is, you need to type **System.out.println**, then an open parenthesis, then the stuff you want to print, then a close parenthesis, then a semicolon.

What is **System.out.println**? I'll explain it now, but I don't expect you to understand all of it. **System** is a prewritten *class* that is provided as part of the Java language. **out** is a *static instance variable* (of the PrintStream class) instantiated in System class. **println** is a *method* of the variable **out**. We've used words like *class*, *instance variable*, and *method*. All of these should be new and wonderfully strange. We will explain these concepts soon enough.

For *now*, all you have to know is that if you want to print, you use **System.out.println**, followed by open parenthesis, the thing you want to print (it can be an expression), and close parenthesis.

The stuff between the parentheses is called the *argument*. This has nothing to do with fighting or arguing. It's just information you are giving to **System.out.println** so it knows what to print.

System.out.print

System.out.println puts a newline character after the argument is printed to the screen. What if you didn't want to do that? Perhaps you wanted to print two pieces of information to the same line.

You should use **System.out.print** instead. This statement prints the argument, but doesn't print an additional newline.

Let's see an example:

```
System.out.print( "Why, " ) ;                    // Using print with a String argument
System.out.print( “23” ) ;                       // Using print with int argument
System.out.println( " is a mighty fine age!" ) ; // Using println
```

The output would be:

Why, 23 is a mighty fine age!

Concatenation and Mixed Types

What happens if you want to print something to the screen, but it has different types? In the previous example, we printed a **String**, an **int**, and another **String**.

Do we need three print statements (a print statement is either of the **print** or **println** variety)?

We don't. That's because string concatenation works with other types than strings. In particular, as long as one of the operands of **+** is a string, Java does string concatenation. This may require creating a string version of the non-string operand.

For example, we can write:

```
System.out.println( "Why, " + age + " is a mighty fine age!" ) ;
```

Due to left associativity, Java evaluates **"Why, " + age** first. This creates a string version of the value of **age**. If **age** is 23, then it produces **"23"**. The concatenation results in a new, temporary string **"Why, 23"**.

"Why, 23" is then concatenated to **" is a mighty fine age!"** which evaluates to yet another new temporary string, **"Why, 23 is a mighty fine age!"**. This is the final evaluated result, which is then given to **System.out.println** to print.

This is an interesting example for several reasons. First, you can put an expression as the argument of **System.out.println**. Second, this expression is evaluated before **System.out.println** prints it out.

Third, and this is not at all obvious, Java can perform some shortcuts to printing. You may not realize it, but each time a concatenation occurs, a new string is created. This is considered slow. It would be better if Java would just create one big string once, made from all the three parts. It can do this. As a programmer, you don't have to care (at least not now). It doesn't hurt to think of Java doing the series of concatenations as we just described, even if, in reality, it may perform this concatenation is a more optimized way.

System.out.println prefers Strings

Even though **System.out.println** (and **print**) can handle any primitive type and **String** expression as argument, you will often want to take advantage of string concatenation.

This means that at least one subexpression has to be a **String**, preferably, the leftmost. One way to guarantee this is to use an empty string at the beginning of the concatenation. For example,

```
System.out.println( "" + age + isTall ) ;
```

where **age** is an **int** and **isTall** is **boolean**. By making **""** the leftmost string being concatenated, we can force the **+** to stand for string concatenation.

Let's see why. **"" + age** does string concatenation since one of the operands is a string (namely **""**). Evaluating this results in a new temporary string (say, **"23"**). This is then concatenated to **isTall**. Since the temporary string is a **String**, we again use string concatenation.

Recall that concatenation means "to put together".

Here's a rule of thumb. Use the empty string trick if the arguments you are using don't work. For the most part, you won't need to use this trick, because you'll print a string in the concatenation somewhere early.

This is the problem with +. It's overloaded. Its meaning depends on context. If we had some special operator such as @ that stood for string concatenation, we'd do even better off because then neither operand would have to be a **String**. It would figure this information from the operator.

Instead, because + is overloaded, Java has to figure out whether it's **int** addition or **double** addition or string concatenation.

Escape Sequences (Outsource: 7-13)

To output nonprinting characters such as a *backspace* or *tab* you must use and **escape sequence**, which always begins with a backslash. The following list describes some common escape sequences that are used in the Java programming language.

Escape Sequence	Description
\b	Backspace
\t	Tab
\n	Newline or linefeed
\f	Form feed
\r	Carriage return
\’	Apostrophe
\”	Quotation mark
\\	Back slash

Comments (Outsource: 2-20)

Program comments are non-executing statements that you add to a program for the purpose of documentation. Programmers use comments to leave notes for themselves and for others who might read their programs in the future. At the very least, your programs should include comments indicating the program’s author, the date, and the program’s name or function.

There are three types of comments in the Java programming language:

- **Line Comments** start with two forward slashes (//) and continue to the end of the current line. Line comments can appear on a line by themselves or at the end of a line following executable code.
- **Block comments** start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/). Block comments can appear on a line by themselves, or a line before executable code, or after executable code. Block comments also can extend across as many lines as needed.
- A special case of block comments are **javadoc** comments. They begin with a forward slash and two asterisks (/**) and end with two asterisks and a forward slash (**/). You can use javadoc comments to generate documentation with a program named javadoc.

Program Grading Process

- All programs are assigned a point value, starting with the lowest and working up to the highest, increasing in difficulty level as you go.
- Work as many programs as you can, working in order from the first to the last.
- Do not skip any program, as each new one introduces a new concept.
- As you finish each one, have your teacher come and grade it from the monitor and check it on your grade sheet.
- Final project grade will be based on the programs successfully checked. For each previous program not graded from the monitor, 10 points will be deducted. For example, if the 90 point version is successfully completed, but the 60, 70, and 80 point versions were skipped, 30 points are deducted, resulting in a grade of 60. Additional points may also be deducted for improper style in source code or improper output format.

EXAMPLE SOURCE CODE

```
**************************source code heading**************************

// <Your Name>
// <Your Class Period>
// <Today's Date>

******************** Program definition - problem statement ********************

/* This is a sample program that demonstrates the format for writing a program.  It
includes seven simple output statements. */

************************ source code example ******************************
public class StandardOutput
{
	public static void main(String args[])  {
		StandardOutput so = new StandardOutput();
		so.output();
	}

	public void output() {
		System.out.println("Ima Kidd");
		System.out.println("1st Period");
		System.out.println("September 6, 2011");
		System.out.println("\n\n");  //creates two blank lines;
		System.out.println("The Falcons are the best!");
		System.out.println("JV Computer Science awesome, baby!");
		System.out.println("\n\n");
	}
}
```

LAB 01 - ASSIGNMENT

Lab 01A - 50 points

OBJECTIVE

WAP (write a program) that displays the output shown below with **one `println`** statement.

OUTPUT

Java is a fine language. Computers are great. Programming is fun!

Lab 01B - 60 points

OBJECTIVE

WAP that displays the output shown below using **three `println`** statements.

OUTPUT

Java is a fine language.
Computers are great.
Programming is fun!

Lab 01C - 70 points

OBJECTIVE

WAP that displays the output shown below using only **one `println`** statement.

OUTPUT

Java is a fine language.
Computers are great.
Programming is fun!

Lab 01D - 80 points

OBJECTIVE

WAP that displays the output shown below. Use as many **`println`** statements as necessary.

OUTPUT

```
JJJJJJJJJ    VV    VV    HH       HH     SSSSS
   JJJ        VV   VV     HH       HH    SS
   JJJ         VV VV      HHHHHHHHH     SSSSS
JJ  JJJ         VVV       HH       HH         SS
 JJJJ             V       HH       HH    SSSSS
```

Lab 01E - 90 points

OBJECTIVE

WAP that displays the output shown below. Use as many **`println`** statements as necessary.

OUTPUT

```
     /\
    /  \
   /    \
  /------\
  |  _   |
  |  | |  |
  |  | |  |
   ------
     **
     **********
```

Lab 01F - 100 points

OBJECTIVE

WAP that displays the output shown below. Use as many **`println`** statements as necessary. Put your name in the ouput where it says *Your Name*.

OUTPUT

```
      r       e
       u     m
      o       a
     Y       N
******************
*                *   *
*    Java        *   *
*                ****
*   is fun       *
*                *
*                *
***************
```