CS201 Lecture Notes, Mock
Overview of Programming

What is programming? Quite simply it is planning or scheduling the performance of some task or event. In the context of computers, computer programming is the planning of a sequence of steps for a computer to follow. Consequently, a computer program is a list of instructions to be performed by a computer.

Writing a program

There is no little man inside the computer. It is a mindless automaton that only does exactly what you tell it to do. So you, the human programmer, must first design a solution to your problem, and then implement it. Here is one lifecycle for designing and implement computer systems:

1. Problem-Solving and Specification Phase
    a. Analysis, Specs of the problem and solution
    b. Design solution, algorithm
    c. Verify solution
2. Implementation Phase
    a. Translation solution into program
    b. Testing
    c. Debugging
3. Maintenance Phase
    a. Use
    b. Maintain, perhaps looping back to step 1

There are many other development methodologies, many of which are discussed in more detail in an Information Systems or Software Engineering course. We will weave some of these concepts into this course as we progress. One of the big dangers of programming is the temptation to jump straight to the implementation. This is dangerous because it may result in a sloppy solution that was not thoroughly designed.

Algorithms to Programs

A key component prior to implementing a computer program is the design of an **algorithm**. An algorithm is simply a sequence of steps to solve a particular program. A recipe for cheesecake is an example of an algorithm (e.g. add ingredients, bake, etc.)

A sample algorithm to calculate someone's wages for the week might look like:

1. Look up employee pay rate
2. Look up hours worked per week
3. Multiply hours * pay rate
4. Calculate overtime wages (perhaps a separate algorithm to do this)
5. Add overtime + basic rate

Once the general solution is developed, the algorithm should be tested mentally or manually, and may need to be redefined. More complicated problems require algorithms to be more carefully designed. Some solutions may be much more efficient than others. Consider searching for a name through a list of names. One could do the search sequentially and be guaranteed to find the name. Given the list of 8 names below, if we want to see if "Wright, Eaton" is in the list using sequential search would require looking at all eight names:

Apple, Bob
Atto, Tom
Attrick, Jerry
DeBanque, Robin
Fresco, Al
Guini, Lynn
Oki, Kerry
Wright, Eaton

But since the names are already sorted, a sequential search is not nearly as efficient as binary search or other methods that can let us jump to the desired name more quickly.

A binary search operates by comparing our target name with the median name in the list. If looking for "Wright, Eaton" we first compare "Wright, Eaton" to the middle name in the list, which is "DeBanque, Robin". Since Wright is alphabetically after DeBanque we know that if Wright is in the list it must come after DeBanque. This means we can throw away any names preceding DeBanque. We are now left with a new list where we can repeat the same process:

Fresco, Al
Guini, Lynn
Oki, Kerry
Wright, Eaton

Now we compare Wright to Guini, narrow the list to "Oki" and "Wright", and finally we will find "Wright". We can find any name in the list in a maximum of four comparisons, an improvement over the eight comparisons in the sequential algorithm! Although this may not sound like a great improvement, if there were millions of names then the efficiency of the binary search algorithm quickly becomes apparent. With n items, binary search performs only $\log_2(n)$ comparisons.

Finally, after the algorithms have been debugged, programs can be constructed. This is the process of converting the English algorithms to a strict set of grammatical rules that are defined by the programming language. There is syntax to the rules as well as semantics. Syntax refers to the order of instructions, like grammatical rules ("favorite the 201 class computer" is syntactically incorrect). Semantics refers to the understanding of the syntax ("green ideas sleep furiously" is syntactically correct but semantically vague).

An incorrect application of either will lead to errors or bugs; the semantic bugs are the most difficult to find.

Once again, sometimes it is tempting to take a shortcut and not spend the time defining the problem and jump straight to coding. At first this saves lots of time, but in many cases this actually takes more time and effort later if the program needs to be redesigned due to mistakes. Developing a general solution before writing the program helps you manage the problem, keep your thoughts straight, and avoid mistakes that can take much longer to debug and maintain than to code. Algorithms are typically developed in **pseudocode**, which is a combination of English and actual programming code

Documentation is another key part of the programming process that is often ignored. Documentation includes written explanations of the problem being solved, the organization of the solution, comments within the program itself, and user manuals that describe how to use the program. You will be given guidelines on how to document your code, and your programs will also be graded on the documentation.

Brief History of Programming Languages

1958: Algol defined, the first high-level structured language with a systematic syntax. Lacked data types. FORTRAN was one of the reasons Algol was invented, as IBM owned FORTRAN and the international committee wanted a new universal language.

1965: Multics – Multiplexed Information and Computing Service. Honeywell mainframe timesharing OS. Precursor to Unix.

1969: Unix – OS for DEC PDP-7, Written in BCPL (Basic Combined Programming Language) and B by Ken Thompson at Bell Labs, with lots of assembly language. You can think of B as being similar to C, but without types (which we will discuss later).

1970: Pascal designated as a successor to Algol, defined by Niklaus Wirth at ETH in Zurich. Very formal, structured, well-defined language.

1970's: Ada programming language developed by Dept. of Defense. Based initially on Pascal. Powerful, but complicated programming language.

1972: Dennis Ritchie at Bell Labs creates C, successor to B, Unix ported to C. "Modern C" was complete by 1973.

1978: Kernighan & Ritchie publish "Programming in C", growth and popularity mirror the growth of Unix systems.

1979: Bjarne Stroustrup at Bell Labs begins work on C++. Note that the name "D" was avoided! C++ was selected as somewhat of a humorous name, since "++" is an operator in the C programming language to increment a value by one. Therefore this name

suggests an enhanced or incremented version of C. C++ contains added features for object-oriented programming and data abstraction.

1983: Various versions of C emerge, and ANSI C work begins.

1989: ANSI and Standard C library. Use of Pascal declining.

1998: ANSI and Standard C++ adopted.

1995: Java goes public, which some people regard as the successor to C++. Java is actually simpler than C++ in many ways, and cleaned up many of the ugly aspects of C++.

Note that this is not a history of all programming languages, only C++ to Java! There are many other languages, procedural and non-procedural, that have followed different paths.

What is a Programming Language – Assemblers, Compilers, Interpreters

Internally, all data is stored in binary digits (bits) as 1's and 0's. This goes for both instructions and data the instructions will operate on. This makes it possible for the computer to process its own instructions as data.

Main memory can typically be treated like a large number of adjacent bytes (one byte is 8 bits). Each byte is addressable and stores data such as numbers, strings of letters, ASCII codes, or machine instructions. When a value needs to be stored that is more than one byte, the computer uses a number of adjacent bytes instead. These are considered to be a single, larger memory location. The boundaries between these locations are not fixed by the hardware but are kept track by the program:

Memory Address	Contents	
…	…	
Byte 4000	11101110	(2 byte memory location at 4000)
Byte 4001	11010110	
Byte 4002	10101011	(1 byte value, e.g. ASCII char)
Byte 4003	11010000	(3 byte memory location at 4003)
Byte 4004	00000000	e.g. string
Byte 4005	11011111	
…	…	

There is only a finite amount of memory available to programs, and someone must manage what data is being stored at what memory address, and what memory addresses are free for use. For example, if a program temporarily needs 1000 bytes to process some data, then we need to know what memory addresses we can use to store this data. One of the nice things about Java is that much of this memory management will be done for us by the Java virtual machine (more on this later).

Computer instructions can be programmed directly as machine code. When computers were first developed, the machine code was the only way to write programs.

Ex: 110110 might be the instruction to add two numbers
110100 might be the instruction to increment a number
etc.

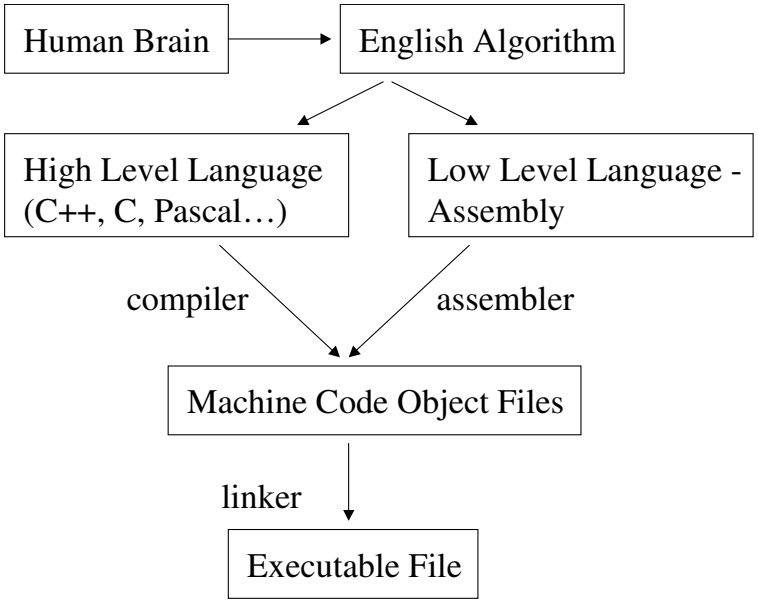
Naturally, it is very tedious to write in direct machine code. One step above machine code is assembly code. Assembly replaces the machine codes with more English-like codes that are easier to remember. These codes are called **mnemonics**.
Ex:

ADD	110110
INC	110100
Etc.	

Although the assembly codes are easier for humans to work with, the computer cannot directly execute the instructions. But people write programs to translate the instructions into machine code. These programs are called assemblers.

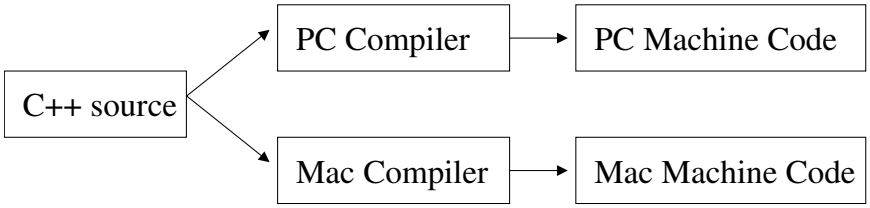
Assembly is still a lot of work for programmers to use because one must know exactly what machine-level instructions are available. Today most programmers use **high-level programming languages** that are easier to use than assembly due to increased generality and a closer correspondence to English and formal languages.

A program called a **compiler** translates programs in high level languages into machine language that can be executed by the computer. C++, C, Pascal, Ada, etc. are all examples of high level languages. (So is Java, but we will get to that in a minute!)



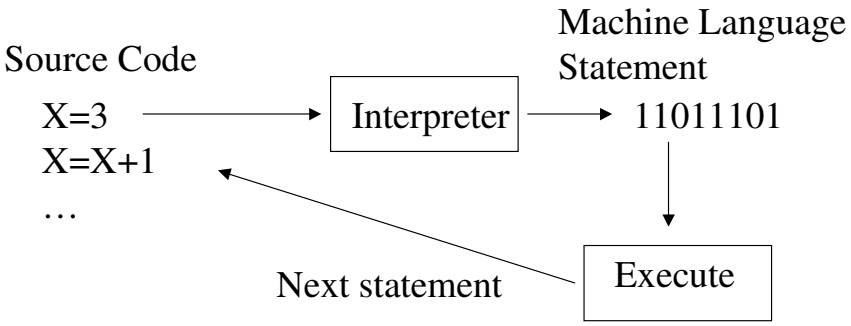
A program written in a high level language is called a **source** program. The compiler takes the source program and typically produces an **object** program – the compiled or machine code version of the source program. If there are multiple source files that make up a final program, these source programs must then be **linked** to produce a final executable.

Note that compilers on different machine architectures must produce different machine code. A Motorola processor (used in older macs) cannot understand machine code intended for an Intel processor. However, if there is a standard version of the high level language, then one could write a program and have it compile on the two different architectures. This is the case for "standard" programs, but any programs that take part of a machine's unique architecture or OS features will typically not compile on another system.



Note that under this model, **compilation** and **execution** are two different processes. During compilation, the compiler program runs and translates source code into machine code and finally into an executable program. The compiler then exits. During execution, the compiled program is loaded from disk into primary memory and then executed.

C++ falls under the compilation/execution model. However, note that some



programming languages fall under the model of **interpretation**. In this mode, compilation and execution are combined into the same step, interpretation. The interpreter reads a single chunk of the source code (usually one statement), compiles the one statement, executes it, then goes back to the source code and fetches the next statement. Examples of some interpreted programming languages include JavaScript, VB Script, some forms of BASIC (not Visual Basic), Lisp, and Prolog.

Question: What happens if you modify the source on a compiled programming language (without recompiling) vs. an interpreted programming language and execute it?

What are the pro's and con's of interpreted vs. compiled?

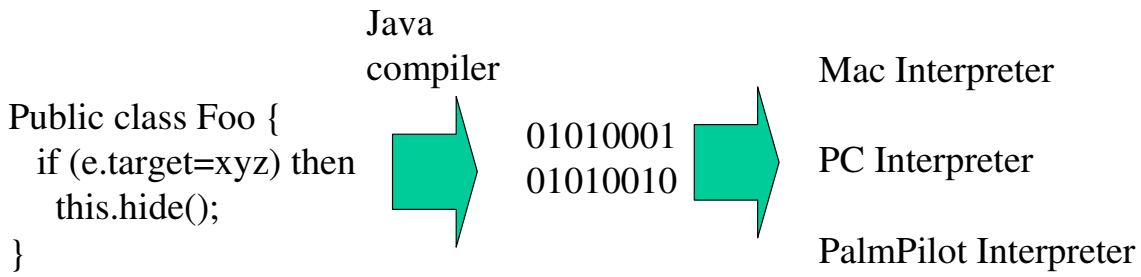
Compiled:

- Runs faster
- Typically has more capabilities
  - Optimize
  - More instructions available
- Best choice for complex, large programs that need to be fast

Interpreted:

- Slower, often easier to develop
- Allows runtime flexibility (e.g. detect fatal errors, portability)
- Some are designed for the web

In the midst of compiled vs. interpreted programming languages is Java. Java is unique in that it is both a compiled and an interpreted language (this is a bit of a simplification with "Just In Time" compilers, but we will ignore that distinction for now). A Java compiler translates source code into machine independent **byte code** that can be executed by the Java **virtual machine**. This machine doesn't actually exist – it is simply a specification of how a machine would operate if it did exist in terms of what machine code it understands. However, the byte code is fairly generic to most computers, making it fairly easy to translate this byte code to actual native machine code. This translation is done by an interpreter that must be written on different architectures that can understand the virtual machine. This interpreter is called the Java Virtual Machine (**JVM**) or Java Runtime Environment.



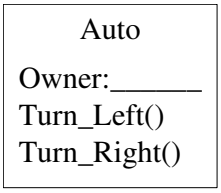
The great benefit of Java is that if someone (e.g. Sun) can write interpreters of java byte code for different platforms, then code can be compiled once and then run on any other type of machine. Unfortunately there is still a bit of variability among Java interpreters, so some programs will operate a bit differently on different platforms. However, the goal is to have a single uniform byte code that can run on any arbitrary type of machine architecture.

Another benefit is we can also control "runaway" code that does things like execute illegal instructions, and better manage memory. We will discuss these later in the course.

Object-Oriented Programming

Java was designed for object-oriented programming, often abbreviated as OOP. OOP is a programming paradigm that uses objects. An **object** encapsulates both data and actions. In other words, an object consists of code that can perform the actions, and data are values to be acted upon. The code is placed in what are called **methods**. If you have used other programming languages, a method is essentially the same thing as a function or procedure. Objects of the same **type** are said to be of the same **class**.

For example, consider a class for an Automobile computer simulation below:



Auto Class

The class can be considered like a template, or blueprint, for automobile objects. This particular class defines a space for the owner of the auto, but since this is just a template it is left blank. Actions we might have for the auto (i.e. the methods) are to turn left or turn right. To complete the definition for the class, we would have to write code that implements what to do when we want to turn left and turn right. If this were a real simulation, we would likely have many more methods defined.

We will discuss how to define our own objects in detail in a few more weeks. However, we will be using predefined objects right away.

Hello, World

The first program that many people write is one that outputs a line of text. In keeping with this vein, we will start with a program that prints, "Hello, world". A similar program is also in the textbook, but I have changed it slightly to demonstrate use of an object. First, here is the program in its entirety.

File: HelloWorld.java

```
/*
 *  Normally you would put your name and assignment info here
 *  This program prints out "Hello, World" and today's date.
 */
import java.util.Date;  // Loads in code to create Date objects

class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world!");
        Date today = new Date();
        System.out.println("Today is " + today.toString());
    }
}
```

Before we try to compile and run this program, let's go through a description of what is in here. First, any part of a line that begins with // is considered a comment and from the // on to the end of the line, the text is ignored. Descriptive text about what the program does should go into the comment fields. An alternate method of writing comments is to use /* and then any text is ignored until */ is encountered. We'll use comments to describe the input, output, and dependency behavior of the program. Also use comments as a place to type your name with the file!

A Java program can be composed of many files. In this case, our program consists of one file. Generally, each file corresponds to a class. A Java program should have only one file that contains the special method called **main**.

After the comments, the first line of HelloWorld.java is "import java.util.Date;". This line of code tells the Java compiler that we'd like access to the Date code (called a **package** in Java) that has been previously written so we don't need to write it ourselves. In this case the Date code defines an object that can do things with dates. If we don't include this line then if we try to use a Date object then the compiler will complain because it won't know where the Date code exists.

Next is the line "class HelloWorld". This defines the name of the class (i.e. the object). **The class name should match the name of the file!** Java is case-sensitive, so Java will complain about a file named "helloworld.java" but the contents of this file contains "HelloWorld" instead. We will place each class object into its own separate file.

The left curly brace is used to denote where the body of the class begins. The right curly brace denotes where the body of the class ends. It is common notation to put the right end brace on a line on its own in the same column as where the class begins, so one can see which curly brace matches with what statement. There is great debate regarding the placement of the curly braces. At this point it is worth mentioning that the compiler doesn't care about **whitespace** between instructions. Whitespace is spaces or carriage returns. You can add as many spaces or blank lines as you wish so that the program is easier for humans to read. You need at least one whitespace character to separate instructions, but others will be ignored. If you wanted to, you could write the entire program on a single line! This would not make it very human-readable, but the compiler will not care.

The next line is "public static void main(String[] args) {". This defines a function called "main" in the HelloWorld object. As indicated earlier, main is a special function. This is where your program begins execution! In Java, functions are often referred to as **methods**. I will use these terms interchangeably. You can think of a method as a collection of code that does some specific task. The main method has a few terms that will look cryptic. You'll learn more about these things later, but for now you can consider this as "boiler plate" that you will just put in all your programs to make them work. Nevertheless, here is a short description of what these terms mean:

public	- The method is made available to anyone that wants to use it
static	- Only make one copy of this method
void	- This method should not return any value (e.g., functions can be designed to return values, f(x))
String[]	- A String refers to a block of ASCII characters like a word or sentence. The [] indicates that we want an array, or a collection, of many strings.
args	- This is a variable name used to refer to the strings In most of our programs we won't use this, but it refers to command-line arguments passed to our program

The next line is "System.out.println("Hello, world!"); ". This entire line is called a **statement**. Every statement must end with the semicolon, also known as a statement terminator. One of the things that gets tricky for beginning programmers is where to put the semicolons; it just takes experience to learn where the semicolons go!

As you might guess, `System.out.println` will output to the screen whatever is contained in double quotes.

The next line "Date today = new Date();". This particular statement declares a **variable** to be of type `Date`. Date is another object that we is already defined by other programmers for us. In other words, we are going to create an instance of this other object. At this point, we are only defining that we would like a Date object, and we will

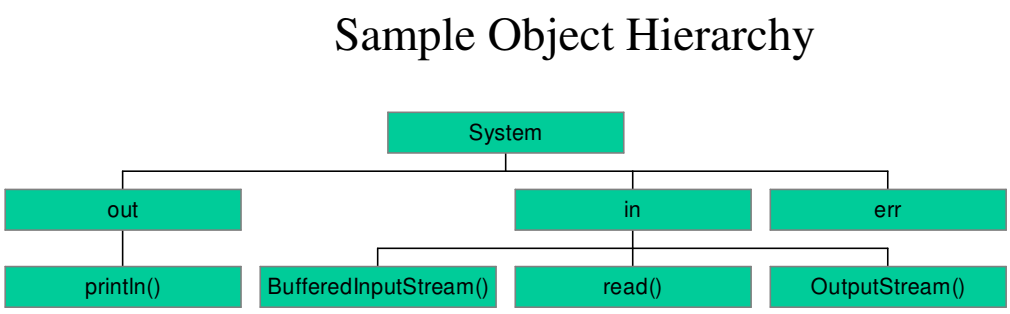
reference this object through the name “today”. The programmer has the luxury of picking whatever name is appropriate to assign to this variable.

A key operator here is the word **new** which tells the Java compiler to allocate the object. This allocates the memory to put this object into memory and also runs any code that might be associated with initializing the object.

The next line “System.out.println("Today is " + today.toString());” which prints out the string “Today is” and then concatenates onto that the value returned by the method `toString()` as defined for the `today Date` object. The parentheses are used to indicate that this is a method or function that we are invoking. In this case, the toString() method computes the current date and returns it as a string which can be output and read. You can think of this return value as taking the place of today.toString() in the program.

The final lines in this file are closing curly braces to denote the end of the main function and the end of the class.

The dotted notation of using periods starts at a high-level Java object, and each dot indicates a more specific Java object. Objects form a hierarchy. For example, here is a small object hierarchy for System:



System.out.println() invokes the method to print a string of data to the screen. Similarly, System.in.read() would invoke a function to read input from the keyboard. Each “dot” moves us down the hierarchy to a more specific function or object.

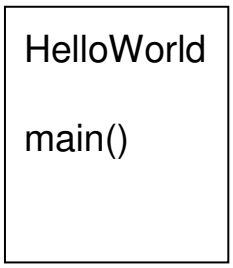
To print a string, note that we used double-quotes. Any double-quoted block of characters is considered a literal string, and is created by concatenating together the appropriate ASCII characters.

Once again, you might not understand everything that is going on here, but it should make more sense as we go along.

Execution

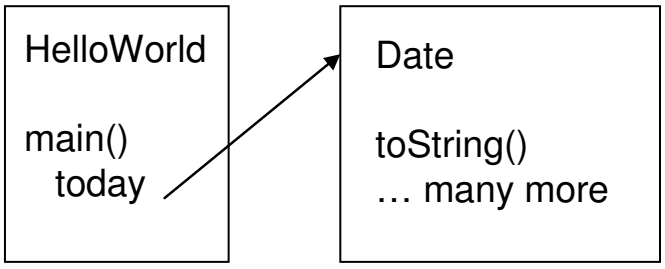
When we first run the program, initially Java will create an object for HelloWorld. This is depicted by the box below, which has a main method:

1.



2. The first line of code: `System.out.println("Hello, world!");`
This outputs "Hello,world!" to the screen.

3. The next line of code: `Date today = new Date();`
This creates a new Date object and we refer to it by the name today:



4. The next line of code: `System.out.println("Today is " + today.toString());`

This invokes the toString() method of the Date object which returns back today's date and time. The value is then concatenated onto "Today is" and returns something like: "Today is Sun Aug 20 22:02:36 AKDT 2008" to the screen.

Compilation

The process of entering and compiling your program is different depending upon what development environment you are using. We'll start by using a most primitive development environment – using a text editor in conjunction with the java command-line compiler. This should give you some idea of what is happening at a direct level. Other development environments include programs that allow more graphical views of your code. These are called IDE's (Integrated Development Environment). We'll use the NetBeans IDE for most of the class. There are also several other IDE's you can download from the java.sun.com web page for free.

To compile your programs using the command-line compiler, first enter the code using a text editor. For example, you could use pico under unix, or Notepad under Windows. The CS lab has a text editor called TextPad that you can use too. Make sure the name of the file matches the name of the class.

After the files have been created, compile them using the javac command. The arguments are the names of the Java source code programs:

- javac HelloWorld.java

If there were any errors, the compiler will complain and tell you it encountered problems. If all goes well, you'll be returned to the prompt and you should now have a file named "HelloWorld.class" in your current directory. These are the compiled versions of the Java source code. To run them, use the Java interpreter:

- java HelloWorld

This will run the program. Note that we didn't enter the ".java" or the ".class" upon executing the program. This command invokes the Java interpreter which will convert the java byte code into native machine code and then run it.

Don't forget that Java is case-sensitive, so make sure upper and lower case letters match.