CSE 11: Lecture 2

- Why programming is fun... but hard
- Writing programs in Java
- Compiling and running Java programs: `javac` and `java`
- Kinds of programming errors
- Variables, identifiers, and declarations
- Integer, floating, character, and boolean types, with literals
- Named constants
- A first look at Object Oriented Programming
- Simple input and output

(Reading: Savitch Ch. 1 and part of Ch. 2)

Why is programming fun?

- According to Fred Brooks, manager of the IBM OS/360 development team in the 1960's (quoting from his 1975 book *The Mythical Man-Month*):
  - First is the sheer joy of making things.
  - Second is the pleasure of making things that are useful to other people.
  - Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning.
  - Fourth is the joy of always learning, which springs from the nonrepeating nature of the task.
  - Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought...
- However...

... and why is it hard?

- Brooks goes on to point out:
  - Not all is delight, however. First, one must perform perfectly... If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work...

    Adjusting to the requirement of perfection is, I think, the most difficult part of learning to program.

    -- Fred Brooks, 1975

- In CSE 11, I hope you have fun programming in Java.

  But you will have to pay attention to the "requirement of perfection":

  Using even a high-level, object-oriented language like Java requires much more precision than a natural language like English

- In CSE 11, programming's "requirements of perfection" will come up in learning details of Java, in meeting specifications of programming assignments, etc.

How to program: 4 steps plus

- 4 steps to keep in mind:
  - Design your program
  - Create one or more source code files that implement your design
  - Compile your source code files
  - Run the program

- Plus: These 4 steps almost never work exactly as planned...
  - So you will need to test the operation of your program, and fix any problems with it
  - This is the process of *testing and debugging*, and it is not separate step as much as it is involved in all 4 steps
  - Debugging can involve redesigning, rewriting, recompiling, rerunning, retesting...

Design your program

- Before creating any source code files, design your program:
  - Understand the requirements, what your program should do
  - Come up with an algorithm (a complete and precise sequence of steps) that will do it
  - Sketch the algorithm in ‘pseudocode’ (a combination of English, mathematical notation, class diagrams, flowcharts, actual Java, etc.)
  - THEN translate the pseudocode to Java
- Especially for more complicated applications, design is essential, and is really the hardest part of the process
- Think about what you’re doing first! If you start by writing source code without designing, you will almost certainly have to *re*write what you’ve done

Create Java source code file(s)

- In Java, any code you write must be contained within a class definition
  - We will talk a lot more about what a class is... Classes are an essential concept in object-oriented programming
  - But for now: just think of a Java class as a capsule that contains Java code
- Normally, you will define one public class per source code file
  - A typical well-designed program defines several public classes, and so will require writing several source code files
  - A well-designed program typically also makes use of pre-existing classes, e.g. from the Java standard library packages
- A source code file must have exactly the same (case-sensitive) name as the public class defined in it, with a "`.java`" extension.
- Create the file and enter its contents using your favorite editor

Compile the Java source code file(s)

- Using the standard Java JDK, you compile a Java source code file by running the **`javac`** compiler, passing it the name of the source code file on the command line

- The javac compiler will compile the source code file you specify, as well as any other needed Java source code files

- This will produce a compiled version of the source code
  - the compiled version consists of Java bytecodes which are the machine language of the Java Virtual Machine (JVM)
  - the compiler puts the bytecode in a file with the same name as the source code file, but with the **`.java`** extension replaced by a **`.class`** extension

Run the program

- Using the standard Java JDK, you run a Java program by running the **`java`** interpreter, passing it on the command line the name of a compiled, bytecode file, but without the "**`.class`**" extension.

- This class name you supply to **`java`** must be the be a class that contains the definition of a **`public static void`** method named **`main`** that takes an array of Strings as argument (more on this later)

- Your Java program then starts executing with the first statement in that **`main`** method

- Any other bytecode files required by your program will be loaded and linked as needed by the Java interpreter

A simple example: Hello world!

- A traditional example when learning a programming language is the simplest possible program: a console application that says “Hello world!”
  ... so, here it is in Java:

```
public class HelloWorld {

   public static void main (String args[]) {

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

   }

}
```

- Now, how to run this program?

A simple example, cont'd.

- The "Hello World" example is only a few lines of code; not much design was needed for this program
- As required, all the Java code is within a class definition; in this case, there is just one class, a public class named **`HelloWorld`**
- The class could have any name that is a legal Java identifier; but once it was selected, its definition has to appear in a file named **`HelloWorld.java`**
- The public class **`HelloWorld`** contains a public static method named **`main`** that takes an array of Strings as argument
- So when **`HelloWorld.java`** is compiled and run, the program will starts executing with the first statement in that **`main`** method
- In this case, there is only one statement there:

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

  which is a call to the **`println`** method of the **`out`** static object contained in the **`System`** class, passing it as argument the String literal **`"Hello World!"`**.

Compiling and running the Hello World program

- Using the standard Java JDK, you compile a Java source code file by running the **`javac`** compiler, passing it the name of the source code file on the command line. In this case:

  ```
  javac HelloWorld.java
  ```

- ... which will produce a compiled version of the source code. The compiler puts the bytecode in a file with the same name as the source code file, but with the **`.java`** extension replaced by a **`.class`** extension. In this case: **`HelloWorld.class`**

- Using the standard Java JDK, you run a Java program by running the **`java`** interpreter, passing it on the command line the name of a compiled, bytecode file, but without the "**`.class`**" extension. In this case:

  ```
  java HelloWorld
  ```

- The Java program then starts executing with the first statement in the **`main`** method of the class defined in that file, with other bytecode files required by your program loaded and linked as needed In this case, the result is to print
  `Hello World!`
  on the terminal screen.

Testing and debugging

- Getting a program to work the way you want takes time and effort... it is usually a miracle if it works right the first time

- Debugging and testing are essential to find and eliminate 3 kinds of errors:
  - Syntax errors
  - Run-time errors
  - Logic errors

Syntax errors

- ✗ detected when the program is compiled ("compile time")
- ✗ are due to violating the syntax (grammar) rules of the language:
  leaving out the semicolon '`;`' at the end of a statement, using an identifier before it has been declared, etc., etc., etc.
- ✗ compiler *errors* indicate strict violations of language syntax (you *have* to fix these);
  compiler *warnings* indicate possible misuse of the language (you *should* fix these)
- ✗ the compiler is real picky about syntax... it can't tell what you *meant* to write

- ✔ Compiler syntax error messages will report the line number of the source code file containing the error... but sometimes this information is not accurate! If you get several error messages, the first one is usually the most accurate; concentrate on fixing it first, then recompile

- ✔ Keep in mind: The compiler knows everything about the syntax of the language you're writing in, and its error messages are there to help you write error-free code

- ✔ (You might say that it is a goal of computer language and compiler design to make as many programming errors as possible detectable automatically by the compiler.... Java is much better than C and C++ in this respect)

Runtime errors

- detected by the operating system when the compiled program is run (“run time”)
- may depend on user input (which cannot be determined at compile-time)
- are due to things like running out of computer memory, dividing by 0, trying to access another user’s file, etc.

Logic errors

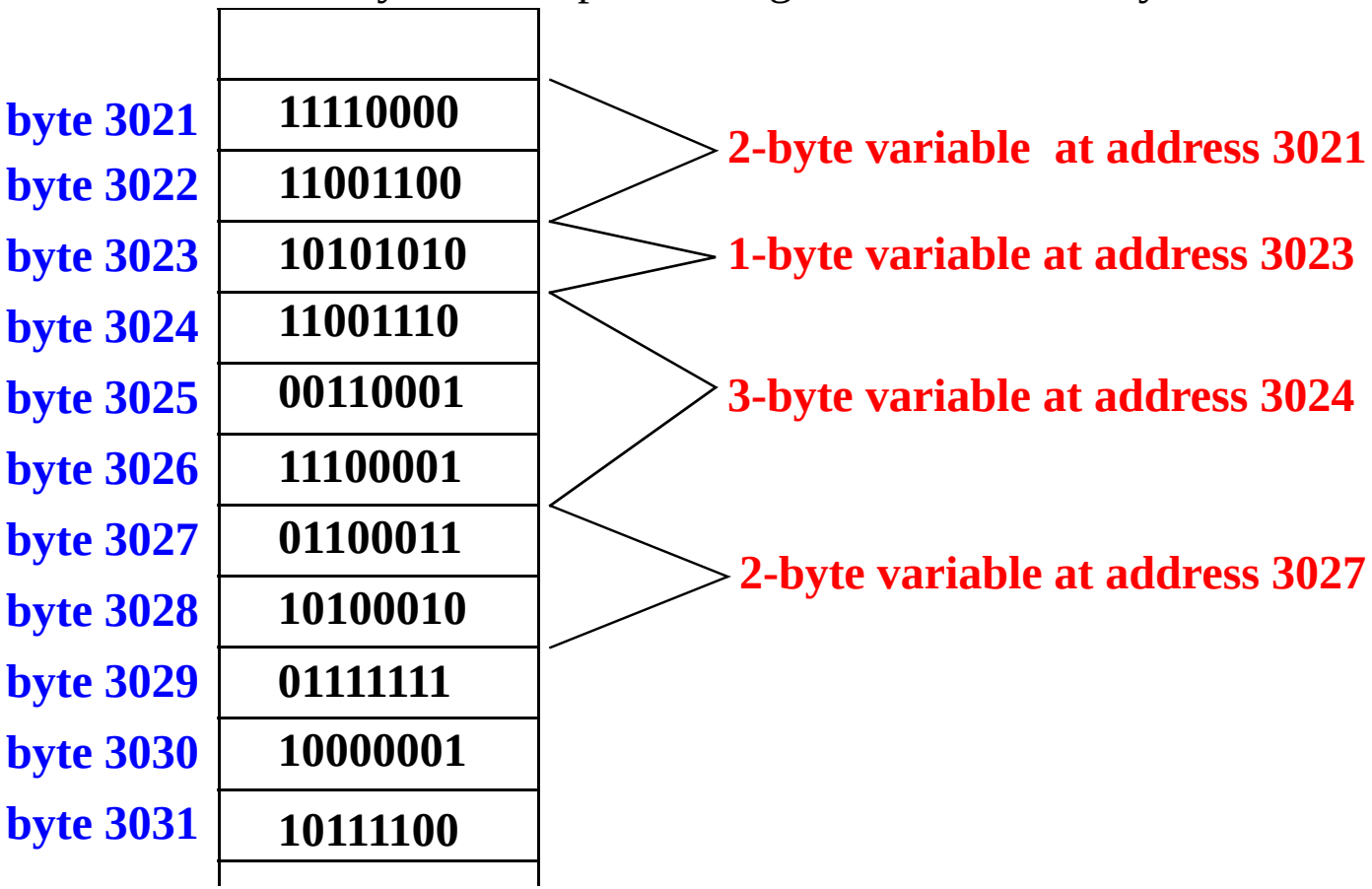
- program compiles and runs without error messages, but...
- program does not do what it was supposed to do!
- can be due to mistakes in design, or in implementation
- Logic errors are the hardest to find, because the computer doesn't generate any error messages for them: extensive program testing and review are necessary

Statements, syntax, semantics, and programs

- A simple program is a sequence of *statements*
  - In a simple Java program like the "Hello World" example, the statements are in the body of the `main` method. You find them enclosed in curly-braces `{}` after the line
    `public static void main(String args[])`
    (that program actually has only one statement:
    `System.out.println("Hello World!");`)
- Each statement in a program has a form, which determines whether it is a legal statement, and if so, what type of statement it is
  - this form is the the *syntax* of the statement
- Each statement in a program tells the computer to do something
  - what it does is called the *semantics* of the statement
- Together, the sequence of statements in the program tells the computer to do what you want done, and that is the goal of programming

Programming, bytes, bits, and variables

- Recall that the memory of a computer is organized into 8-bit bytes



- Programming a computer basically involves getting these 1's and 0's to be what you want. In a high-level language like Java, doing this requires understanding how variables work

What's a variable?

- In computer science, a variable is a piece of computer memory that contains information. Very useful!
- The information in a variable can typically be accessed (read), and changed (written)
- The information stored in a variable is called the value of the variable; it is stored as a pattern of bits (1's and 0's)
- The number of different possible values of a variable depends on how many bits of memory the variable contains...

  1 bit: 2 possible values
  2 bits: 4 possible values
  8 bits: 256 possible values
  16 bits: about 64,000 possible values
  32 bits: about 4 billion possible values
  ...
  n bits: 2^n possible values

- But what does a pattern of bits *mean*?

Java variable declarations: Syntax and semantics

- How the pattern of 1's and 0's in a variable is interpreted in a program (as an integer, fraction, letter, memory address, etc.) depends on the *data type* of the variable

- How you refer to a variable to access or change its value depends on the *name* of the variable

- Java is a strongly typed language:
  - in Java, every variable has a definite type which is determined at compile time
  - in Java, the type of each variable must be declared in your program

- Syntax of a Java variable declaration:

```
typename identifier;
```

  for example,

```
int j;
```

  Semantics: A variable declaration associates a identifier with a variable of the given type, and allocates memory for the variable if it is a primitive type. (In this example, **`j`** is declared to be the name of a variable of type **`int`**.)

Declaring more than one variable in a declaration

- A declaration of the form

  ```
  typename identifier1, identifier2, ... ;
  ```

  for example

  ```
  double foo, bar, baz;
  ```

  associates names with several variables of a certain type, and allocates storage for them if it is a primitive type. Here **`foo`**, **`bar`**, and **`baz`** are declared to be names of variables of type **`double`**.

- Although this multiple-variable form is allowed, it is usually considered clearer, and so better style, to use single-variable declarations

- Next question: What can be an identifier?

Identifier syntax

- Any computer language has rules for what are legal identifiers: the syntax of identifiers
- In Java, an identifier ...
  - begins with a letter or an underscore _
  - contains any number of letters, digits, and underscores _
  - Java identifiers are "case sensitive": `A` is different from `a`, etc.
  - (`$` can appear anywhere in an identifier, but this is discouraged in user programs)
- Which of these are legal Java identifiers?

```
four

Four

4

$4

my 1st favorite object

my_1st_favorite_object

my1stFavoriteObject

O.K.
```

Selecting identifiers

- Identifiers are used for the names of variables (and other things, such as classes, interfaces, packages, and methods)

- Java programmers use this convention:
  - make variable and method names start with a lowercase letter; make class and interface names start with an uppercase letter

- identifiers should be informative --- this helps to make programs easier to understand
  - `distance, rate, time` are usually better identifiers than `e, f, g`

- some identifiers are already reserved by Java and cannot be used as names of variables, classes, or methods in user programs
  - "key words" like `int, double, class` and many others
  - boolean literals `true, false` and the reference literal `null`

- some other identifiers are used as the names of classes and methods in standard Java libraries, and may be confusing to use as the names of classes or methods in user programs; it's best to avoid these as well

Java keywords

- These identifiers are keywords in the Java language; they are reserved, and cannot be used as the names of variables in Java programs:

`abstract`	`default`	`if`	`private`	`this`
`boolean`	`do`	`implements`	`protected`	`throw`
`break`	`double`	`import`	`public`	`throws`
`byte`	`else`	`instanceof`	`return`	`transient`
`case`	`extends`	`int`	`short`	`try`
`catch`	`final`	`interface`	`static`	`void`
`char`	`finally`	`long`	`strictfp`	`volatile`
`class`	`float`	`native`	`super`	`while`
`const`	`for`	`new`	`switch`	
`continue`	`goto`	`package`	`synchronized`	

Types in Java

- In a Java declaration, the *typename* is an identifier that names...
  - ... a reference type, possibly user-defined
    - (we will learn how to do this later); or
  - ... a primitive type
    - (we will learn about all of these right now)

Primitive data types in Java

- Numeric or arithmetic types: for representing numbers
  - *integral* (or *integer*) types: `byte` `short` `int` `long`
    - for representing whole integer numbers
  - *floating* (or *floating-point*) types: `float` `double`
    - for representing fractional numbers, or very large numbers
- *character* type (often considered to be an integral type): `char`
  - for representing letters and symbols
- *boolean* type: `boolean`
  - for representing boolean (true or false) values

Integral types

- ✗ **`byte`**
  - 8 bits of memory each
  - can represent integers in the range -128 through 127
- ✗ **`short`**
  - 16 bits of memory each
  - can represent integers in the range -32768 through 32767
- ✗ **`int`**
  - 32 bits of memory each
  - can represent integers in the range -2^{31} through $2^{31} - 1$
- ✗ **`long`**
  - 64 bits of memory each
  - can represent integers in the range -2^{63} through $2^{63} - 1$

✔ In CSE 11, we'll mainly use **`int`**

Integer literal constants

- Integer literals are used to specify particular integer values in your programs
- Specify an integer value as a string of base-10 digits... preceded by a '-' sign if you want it to be negative
- End with the letter "L" if you want it to be of type **`long`**... otherwise, it will be **`int`**

```
43                                  // Okay!
12345
-22
3000000000L

3,456,789                           // Nope!
3.14159
```

- You can also use base-8 (octal) or base-16 (hexadecimal) integer literals:
  - octal literals start with a 0:

    ```
    073      // octal 73, equal to decimal 59
    ```

  - hexadecimal literals start with 0x:

    ```
    0x3a     // hex 3a, equal to decimal 58
    ```

(In base-16: **`a`** or **`A`** = 10, **`b`** or **`B`** = 11, **`c`** or **`C`** = 12, **`d`** or **`D`** = 13, **`e`** or **`E`** = 14, **`f`** or **`F`** = 15)

Floating-point types

- `float`
  - 32 bits of storage
  - can represent about 9 significant decimal figures
  - largest magnitude: about 10^{38}
  - smallest nonzero magnitude: about 10^{-45}

- `double`
  - 64 bits of storage
  - can represent about 18 significant decimal figures
  - largest magnitude: about 10^{308}
  - smallest nonzero magnitude: about 10^{-324}

Floating literal constants

- this is how to specify a particular floating-point value in your programs
- Specify a floating-point value ...
  - either with a string of digits containing a decimal point, preceded by a '-' sign if you want it to be negative
  - or with a *mantissa*, followed by `e` or `E`, followed by an *exponent* *("engineering notation")*
  - End with the letter "F" if you want type `float`... otherwise, it will be `double`

```
3.14159
-000.000001F
2.
2.0
2.0000
```

`1e3` **represents** 1×10^3 **same as** `1000.`

`-1e-6` **represents** -1×10^{-6} **same as** `-0.000001`

`3.14159E0` **represents** 3.14159×10^0 **same as** `3.14159`

Character type

- **`char`**
  - an unsigned 16-bit integral type
  - can represent any Unicode character, which include the ASCII characters
- Character literals:
  - enclose a character in single quotes to represent it as a **`char`** type literal

    `'a'` the character a
    `'A'` the character A
    `'%'` the character %
  - "escape" characters ...

    `'\n'` the newline, or carriage return character
    `'\t'` the tab character
    `'\b'` the backspace character
    `'\''` the single-quote `'`
    `'\"'` the double-quote `"`
- String literals (Strings are not primitive types in Java, but they are implemented using arrays of chars)
  - a sequence of characters enclosed in double quotes

```
"this is !@#$%^&*( a kind of \n STRING"
```

Boolean type

- `boolean`
  - a full 8 bit byte is used to store a boolean in Java, though 1 bit would suffice
  - can represent only the values “true” or “false”

- Boolean literals
  - there are only two boolean literals in Java:
    `true`
    `false`

Declaring and initializing variables

- In Java, a variable must be declared before it it used at all
- A variable must also be *initialized* (be given a definite value) before its value is used (otherwise javac will give you a compiler error, preventing the risk of "garbage in, garbage out")
- One way to do this is to initialize the variable in a declaration of the form

  typename identifier = value;

  for example,

  ```
  int j = 3;
  ```

  Semantics: declares `j` to be an integer variable, and initializes it to have the value 3.
- Several variables can be initialized in the same declaration

  ```
  double foo = 3.14159, bar, baz = 22.2e22;
  ```
- You can also initialize the value of a variable using an assignment statement; and some variables are automatically initialized for you (we'll talk about these later)

Declaring and defining named constants

- A variable can have its value changed while the program is running
- A named constant can only be initialized: its value cannot be changed
- In Java, a constant declaration-and-definition has the form

  ```
  final typename identifier = value;
  ```

  for example,

  ```
  final double NORMALTEMP = 98.6;
  ```

  declares and defines `NORMALTEMP` to be the name of a double constant with value 98.6.

- Named constants are convenient for giving names to values that should not change during the running of your program ... with a good choice of name, this can make your program more readable and maintainable
- Convention: The names of named constants should be ALL CAPS.

Static named constants

- If you want to create a named constant that will be used only by one method in a class (for example, the `public static void main()` method), put a declaration like

  ```
  final double NORMALTEMP = 98.6;
  ```

  in the body of the method.

- If you want to create a named constant to be used by any method in a class, and even outside the class, put a declaration like

  ```
  public static final double NORMALTEMP = 98.6;
  ```

  in the class, outside the body of any method.

  ... Example: The standard Java library class `Math` defines a public static final double constant `PI`, so you can refer to it as `Math.PI`

- This gets us into some aspects of Object Oriented Programming... which we'll continue with now

Object oriented programming: classes, objects, methods

- Java is an object-oriented programming language
- In object-oriented programming, there are *classes* and *objects*
- Classes are data types; objects are variables of those types
  - In Java, classes are treated differently from primitive types, and we will talk a lot about these differences
- To perform an "object-oriented" action, you invoke a *method* of a class or object
  - if the method is associated with a class, it is called a "static method"
  - if the method is associated with an object, it is called an "instance method"
- We will talk a lot more about this whole idea, but for now, we will consider how to use methods for text terminal I/O in Java

Printing things on the terminal screen: using the System.out object

- To print something to the screen, invoke the **`print`** instance method of the **`out`** object in contained in the **`System`** class with a statement of this form:

```
System.out.print(expression);
```

  - Semantics: computes the value of the expression, and prints the value on the terminal screen

- ... or, use the **`println`** instance method:

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

  - Semantics: computes the value of the expression, and prints the value on the terminal screen, followed by a newline/carriage return

  - Note that **`"\n"`** in a string literal, or the char literal **`'\n'`** print as a newline

Getting things from the keyboard: the SavitchIn class

- To get values typed in from the keyboard, in CSE 11 you can use public static methods of the **`SavitchIn`** class

- Note: The **`SavitchIn`** class is not a standard part of the Java language...
  ... using standard parts of the Java language to do keyboard input is slightly complicated, so we'll use **`SavitchIn`**'s methods instead to make it simpler

- The source file **`SavitchIn.java`** is available from your class account. By the end of CSE 11, you should understand everything that's in there!

Using SavitchIn: some static methods of the SavitchIn class

- **`SavitchIn.readLineInt()`** reads an integer literal constant typed in at the keyboard, and returns its value as an int. The literal should be the only thing typed on a line of input...
  when **`int i = SavitchIn.readLineInt();`** is executed, if the user types
  `3145 <enter>`
  then **`i`** gets the value **`3145`**
- **`SavitchIn.readInt()`** reads the next whitespace-delimited integer literal constant available in the keyboard input, and returns its value as an int... This is useful if several literals are typed on one line.
- **`SavitchIn.readLineDouble()`** reads a floating literal constant typed in at the keyboard, and returns its value as a double. The literal should be the only thing typed on a line of input
- **`SavitchIn.readDouble()`** reads the next whitespace-delimited floating literal constant available in the keyboard input, and returns its value as a double.
- **`SavitchIn.readLine()`** reads a line of input typed in at the keyboard, and returns its value as a String object (not including the newline character).
- **`SavitchIn.readChar()`** reads the next single character available in the keyboard input, and returns its value as a char.

Next time

- ✔ Numeric operators and the String concatenation operator
- ✔ The assignment operator and assignment statements
- ✔ Programming and logic
- ✔ Comparison operators and boolean operators
- ✔ Conditional statements
- ✔ Iteration
- ✔ Methods: parameters and return values

(Reading: Savitch, parts of Ch. 2 , 3, and 4)