Announcements

- Course web: http://www.cs.stonybrook.edu/~alee/cse114/

- Survey form: See Announcements on the course web
  - Do it as soon as possible

- Reading assignment for this week: Chapter 1 of Liang
  - You may skip section 1.11

- If you are asked to move to CSE 101, please do so ASAP

- Lab on Thursday, 8/30, is for Windows users only
  - Starting next week everyone will come to every lab session
What is computer science?

- A very broad definition:
  
  Computer science (CS) is the systematic study of computing systems and computation

- Computer science is NOT just programming!

- Programming is an important part of CS
  
  We’re going to learn to program in Java but much of what you will learn are fundamental CS ideas that apply to most any programming language
What do computer scientists do?

Some examples of what computer scientists work on:

- Design computers and their components
- Programming languages
- Operating systems
- Artificial intelligence
- Databases
- Networking
- Theory of computation
- Computer graphics
- Robotics
- Biocomputing
- Many more . . .
What is programming?

- Programming is the process of “giving” instructions to a computer or the computer’s central processing unit (CPU)
  - Also referred to as "writing code" or simply "coding"

- Learning to program is similar to learning a "natural" language like Spanish. You have to learn the:
  - **Syntax**: the words and rules in the language
  - **Semantics**: the meaning of each word or phrase in the language

- A computer is pretty dumb though. . .
  - Consequently, you have to give super clear and precise instructions
  - A computer will happily do the same thing forever (in an infinite loop) if you tell it to do so, even if you didn't mean to!

- Unlike natural languages, programming languages are extremely picky – rules can’t be violated
What is a program?

- **Program**: A sequence of instructions to be carried out by a computer (to perform a computational task)

- **Program execution**: The act of carrying out the instructions contained in a program

- Example: see Hello.java
  - What is the computational task that we perform in this example?
public class Hello {

    public static void main (String[] args) {
        System.out.println("Welcome to CS!!!");
        System.out.println("Let's have some fun.");
    }

}
Low-level computer instructions

- **Machine language**
  - Binary instruction (1's and 0's)
  - Most instructions just move data around or perform simple arithmetic operations

- Example: on Intel x86 processors:
  - 1011000001100001 means to copy a 97 to a particular register

- Binary programming is really hard and tedious but early programmers did exactly this!
“Mid-level” computer instructions

- **Assembly language**
  - Symbolic (meaningful) names for binary instructions and memory
  - A little more readable
  - Feasible for programmers to use

- **Example:**

  ```
  ADD DR, SR1, SR2 ; DR <- (SR1) + (SR2)
  LD DR, LABEL ; DR <= Mem[LABEL]
  LDR DR, BaseR, Offset ; DR <- Mem[BaseR + Offset]
  STI SR, LABEL ; Mem[Mem[LABEL]] <= SR
  ```
High-level computer instructions

- High-level language
  - Symbolic names for assembly instructions and memory
  - Symbolic names for basic operations such as looping
  - Symbolic “shorthand” for a group of instructions
  - Close to "natural" languages - close to being readable!

- Example:
  ```
  print("Welcome to CS!");
  y = a * x + b;
  ```
Why use a high-level language?

- **Conciseness**
  - high-level programming languages allow us to express common operations in a concise and readable fashion

- **Maintainability**
  - Modifying and maintaining code is much easier when the code is concise and easy to read (as compared to lengthy and difficult to read assembly or binary code)

- **Portability**
  - Different CPU's accept different binary instructions
  - Writing in a high-level language allows code to be translated or "compiled" into a platform-specific binary code
  - Allows your code to be "ported" to another platform
Some modern languages

- **Procedural languages**: programs are a series of commands
  - *Pascal* (1970): designed for education
  - *C* (1972): low-level operating systems and device drivers

- **Functional languages**: functions map inputs to outputs
  - *Lisp* (1958) / *Scheme* (1975)
  - *ML* (1973)
  - *Haskell* (1990)

- **Object-oriented languages**: programs use interacting “objects”
  - *Smalltalk* (1980): first major object-oriented language
  - *C++* (1979): “object-oriented” improvement to C
  - *Java* (1995): designed for embedded systems, servers
    - Runs on many platforms (Windows, Mac, Linux, cell phones, . . .)
  - *Python* (1989): general purpose, procedural and objects
  - *Javascript* (~1995): the language of the web, part of every browser
Why Java?

- Java is a straightforward and powerful object-oriented language
  - Simpler than C++
  - Highly productive language

- Platform independent (Mac, Windows, ...)
  - “Write once, run everywhere”

- Java is used heavily in industry
  - Many major companies have utilized Java in their internal technical infrastructure

- Java has good support
  - Many useful prewritten “packages” available as part of the Java ecosystem such as graphics, multimedia, networking, etc.

- The first programming language taught at many universities
Java’s roots

- C was developed in the early 1970’s
  - C was designed to be small, fast, with little built-in safety

- C++ was developed in the late 1970’s
  - C++ is a superset of C and extends C to include object-oriented concepts

- Java borrowed from C/C++ but more concerned about safety and productivity at the cost of some speed
  - Java's creator James Gosling has described Java as "C++ without the guns, clubs and knives"
  - Designed to be used in the Internet era
Java is C++ without the guns, clubs and knives.

— James Gosling —

https://www.azquotes.com/quote/596302
Java Introduction
Ex: Hello program (**Hello.java**)

```java
public class Hello {

    public static void main (String[] args) {
        System.out.println("Welcome to CS!!!");
        System.out.println("Let's have some fun.");
    }
}
```

- Black parts are given by Java
- **Blue** parts are what’s added as my program
- Note some things that are intuitive (?)
  - Indentation is optional (make code more readable)
  - Matching ( and )
  - Matching { and }
  - Matching [ and ]
The coding process

1. Create/Edit source code (.java files)
   Use a text editor to write or edit your code (instructions)
   E.g., emacs, but not Word
2. Compile
   The Java compiler creates bytecodes – an intermediate machine independent
   instruction set that targets a Virtual Machine
   The javac command
3. Run on the Java Virtual Machine (JVM)
   The java command
4. Debug your application

- Repeat steps 1 – 4 as necessary
Java programming model

source code -> text files (*.java)

Java compiler

Platform independent bytecodes

Java Virtual Machine (JVM)

<table>
<thead>
<tr>
<th>JVM for MacOS</th>
<th>JVM for Windows</th>
<th>JVM for Linux</th>
<th>JVM for Android</th>
<th>JVM for ???</th>
</tr>
</thead>
</table>
Traditional programming model

source code

MacOS compiler/linker

MacOS executable

Windows compiler/linker

Windows executable

Linux compiler/linker

Linux executable

text files

(*.c, *.cpp, etc...)
Summary – bytecodes and the JVM

- In order to “write once, run everywhere”, Java uses the notation of a secure universal “Java Virtual Machine” (JVM)

- In Java you do not have to worry about cross-platform idiosyncrasies. . . your program will run “on” the JVM instead

- The Java compiler only has to worry about producing one “binary” known as bytecodes

- The JVM runs your program by interpreting or translating your program’s bytecodes into platform specific machine code

- There are multiple JVM’s for each supported platform (Mac, Windows, Linux, Android, etc.) but they all understand bytecodes
Editing code

- Programmers use a text editor to write code
  - Don’t use a word processor such as MS Word, Google Docs, etc.
  - Don’t use the default PC/Mac editor

- The classic (free): emacs, vim
  - Both are cross-platform (PC, Mac, Linux), keyboard-centric (minimal GUI)

- Standalone GUI-based editors (free):
  - PC: Notepad++, Programmer’s Motepad
  - Mac: TextWrangler, Textmate2
  - Both: Sublime Text 2 (free trial), Atom (free from github)

- You only need one editor that you like!

- Text editors do not know how to compile your code!

- Text editor in IDE’s (next slide)
Java IDE’s

• IDE – Integrated Development Environment
  • Combines text editor, compiler, debugger into one GUI-based environment

• There are several reasonable IDE’s for Java
  • Eclipse (PC, Mac, Linux)
  • Netbeans (PC, Mac)

• We’re going to use/support Eclipse

• Installation instructions are on the course web
Using Eclipse

- In Eclipse, your code must live “in” a Java Project
  - Each Java “project” is a subfolder

- Eclipse wants to manage a “workspace” directory/folder

- Do NOT move or change Eclipse’s workspace manually!
Syntax error example

```java
public class Hello {
    public static void main (String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- Compiler output:
  ```
  Hello.java:2: error: <identifier> expected
  public static void main (String[] args) {
    ^
  1 error
  ```
- The compiler shows the line number where it found the error
- The error message can be tough to understand!
- Takes a little practice to “parse” the error message
A Java program
System.out.println

• A statement that prints a line of output on the console

• Two ways to use System.out.println:
  
  • System.out.println("text");
    Prints the given message as output

  • System.out.println();
    Prints a blank line of output
Java program structure: `class`

```java
// comments about the class
public class MyClass {
    // class body
    // - starts with a left brace {
    // - ends with a right brace }
    // This line is the class header
    // - must match the file name
    // - convention is to capitalize 1st letter of each word
}
```
Java program structure: the `main` method

```java
public class MyClass {
    public static void main (String[] args) {
        statement1;
        statement2;
        ...
    }
}
```

Let's not worry about:
- `public`, `class`
- `static`, `void`
- `String[]` `args`

Think of them as necessary decoration for now.

- method name
- method header
- method body is a block of statements
  - starts with a left brace `{`
  - ends with a right brace `}`
public class MyClass {
    public static void main (String[] args) {
        statement1;
        statement2;
        ...
    }
}

class: a program

method: a named group of statements

statement: a command to be executed
Ex: Hello program (Hello.java)

```java
public class Hello {

    public static void main (String[] args) {
        System.out.println("Welcome to CS!!!");
        System.out.println("Let's have some fun.");
    }
}
```
Names and identifiers

• You must give your program a name

    public class MyProgram { . . . }

• Class naming convention: capitalize each word, e.g., MyProgram

• Your program’s file name must match exactly (MyProgram.java)
  • Includes capitalization (Java is “case-sensitive”)

• identifier: A name given to an item in your program
  • Must start with a letter or _ or $
  • Subsequent characters can be any of those or a number
    • Legal: _myName  TheCure  ANSWER_IS_23  $blind$
    • Illegal: me+u       49ers       side-swipe       Ph.D.’s
keywords

- **keyword**: an identifier that you cannot use because it already has a reserved meaning in Java

    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
Comments

- **comment**: a note written in source code by the programmer to describe or clarify the code
  - Comments are **not** executed when your program runs

- Syntax:
  
  // comment text, on one line, stops at end-of-line
  
  Or
  
  /* comment text, stops at --> */

- Examples:
  
  // This is a one-line comment.

  /* This is a long
   *     multi-line comment.
  */
Using comments

- Add comments
  - At the top of each file (general description, your name, etc.)
  - At the start of every method (seen later) if you think it’s necessary
  - To explain complex pieces of code

Comments are useful for:
- Understanding larger, more complex programs
- Multiple programmers working together, who must understand each other’s code
Comments example

/* Suzy Student, CSE 114, Fall 2018
 * This prints what to do in the US presidential election.
 */

public class Who2Elect {
    public static void main(String[] args) {
        // for H.C.
        System.out.println("If you like xxx");
        System.out.println("Vote for H.C.");

        // for D.T.
        System.out.println("If you like yyy");
        System.out.println("Vote for D.T.");
    }
}

Strings

- **string**: a sequence of characters to be printed
  - Starts and ends with a quote ("" character
  - The quotes do not appear in the output
  - Examples:
    "hello"
    "hi there!"
    "this is a string. It’s sorta long..."

- **Restrictions**
  - May not span multiple lines
    "This is not a legal string."
  - May not contain a “ character
    "This is not a “ legal “ string either."
Syntax errors

- **syntax**: the set of legal structures and commands that can be used in a particular language, e.g.,
  - Every basic Java statement ends with a semicolon (;
  - The contents of a class or method occur between { and }

- **syntax error (compiler error)**: a problem in the structure of a program that causes the compiler to fail
  - Missing semicolon
  - Too many or two few or unmatched braces { }{}
  - Illegal identifier for class name, method name, variable name, etc.
  - Class and file names do not match
  - etc.
Statement blocks

• The left brace “{“ and the right brace “}” are used to group together a block of statements:

```java
{  
    statement1;
    statement2;
    more statements. . .
  }
```

• Often, we’ll give a statement block a name so we can refer to it later. . . a sort of shorthand notation!

• A block of statements is also called a *compound statement*

• This shorthand name is the idea behind *functions* or *methods*. 