CS 111 Summary

User Environment

Terminal window: Prompts user for commands.

BASH shell tips:

- <tab> for command line completion.
- <up>/<down> to step backward/forward through command history.
- !<start of command> will re-execute most recent command matching start
- <left>/<right>/<Ctrl-a>/<Ctrl-e> to navigate and edit line

Common UNIX commands:

- man <command> – get help for a command
- apropos <keyword> – get help for a keyword
- ls – list current directory contents
- cd <directory> – change directory
- mkdir <directory> – make directory
- rmdir <directory> – remove directory
- more/less <file> – show contents of text file
- mv <file> <new file location/name> – move/rename file/directory
- cp <file> <new file location/name> – copy file
- rm <file> – remove file
- yppasswd – change password

Common applications (append "<space>&" in BASH shell for long running applications)

- firefox – web browser
- eclipse – Java integrated development environment (IDE)
- java / javac – Java virtual machine / compiler

Java Application Process

1. Edit a plain-text source code file <classname>.java using a text editor (e.g. gedit, emacs, vim, notepad, text edit, etc.) or an integrated development environment (IDE).
2. Compile (translate) source code <classname>.java into Java bytecode class file <classname>.class by entering the command: javac <classname>.java
3. Interpret (translate and execute) Java bytecode class file with the Java Virtual Machine (JVM) interpreter by entering the command: java <classname>

A Simple Java Application

In file Hello.java:

```
public class Hello {

	public static void main(String[] args) {

		// insert program code here, e.g.:
		System.out.println("Hello, world!");

	}

}
```

Note: The class name after `public class` has to match the filename before the ".java" extension.

Printing

- To print a line to standard output: `System.out.println(`something to print`);`
- A String is a string/sequence of characters. We can create a Java String by enclosing characters between double quotes, e.g. `"Hello, world!"`.
- The backslash (`\`) is the *escape character* for Java Strings. Anything after the backslash is given special meaning in a String.
- A double-quote may be included in a String by placing a backslash (`\`) before it.
- A backslash may be included in a String by using two backslashes.
- There are other useful backslash sequences as well, e.g. newline (`\n`) and tab(`\t`).
- To print to standard output without starting a new line at the end, use `System.out.print(`something to print`);`
- To print a formatted String, use `System.out.printf(`format String, value1, …, valueN`);`
- Format Strings contain formatting placeholders (e.g. %s for String, %d for integer, %f for floating point number) that have print representations of following values substituted in place.

Input

- To create a Scanner object to read from standard input: `Scanner in = new Scanner(System.in);`
- To read an integer from this Scanner object: `in.nextInt()`
- To read a double (double precision floating point number) from this Scanner object: `in.nextDouble()`
- To read a line of characters as a String from this Scanner object: `in.nextLine()`

Variables

- A *variable* is a named place in computer memory to store data.
- A variable declaration includes a data type, a variable name, and, optionally an assignment to that variable (an equal sign "=" followed by a value of the given type) , and ending with a semicolon:
  `Type` `Variable Name` = `Initial Value` ;
- Common variable types: int (integer), double (double-precision floating-point), boolean (true/false), String (an object containing a sequence of characters)
- Java convention is for variable to begin lowercase and continue camelCase.

Examples:

```
double distance = 5.5;
int milesPerGallon = 35;
String greeting = "Hello";
```

Arithmetic / Logic

- Parentheses may be used to clarify or change operator precedence. Complex expressions are evaluated from left to right, from inside to outside.
- Arithmetic:
  - Operators: addition (+), subtraction (–), multiplication (*), division (/), modulus (%)
  - For integers a and b, a / b and a % b are the quotient and remainder of the integer division, respectively.
  - For floating-point division, the numerator and/or the denominator must be a floating-point number. For example, 8 / 5 is 1, but 8.0 / 5, 8 / 5.0, 8.0 / 5.0, `(double) 8 / 5`, etc. are all 1.6.
  - Since the form variable = variable operator value is so common, there are common shorthand forms for arithmetic operators: `i += 1;` is equivalent to `i = i + 1;` and `i -= 1;` is equivalent to `i = i - 1;`
  - `i++` and `i--` are post-increment and post-decrement expressions, that increment and decrement `i`, respectively, but take on the value of `i` *before* the operation.
  - `++i` and `--i` are pre-increment and pre-decrement expressions, that increment and decrement `i`, respectively, but take on the value of `i` *after* the operation.
  -
- Boolean Logic:
  - Equality Operators: equals (==), not equals (!=)
  - Relational Operators: greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=)
  - Logical Operators: and (&&), or (||)
  - Short-circuiting evaluation: In left-to-right evaluation of a logical expression exp1 && exp2 && …, the evaluation ceases at the first subexpression found to be false, since the

value of the total expression is now determined. Similarly, `||` evaluation ceases with the first true subexpression evaluation.

Examples:

```
int numerator = 10 + 2 * (3 - 4);
int denominator = 5;
int quotient = numerator / denominator;
int remainder = numerator % denominator;
double floatDiv = (double) numerator / denominator;


boolean shouldBeTrue
  = (numerator == denominator * quotient + remainder)
    && ((double) quotient != floatDiv);
int a = 1;
int b = 2;
shouldBeTrue = !shouldBeTrue || a > b || a >= b || a == b
               || a <= b || a < b || a != b;
```

Objects

- An *object* is a collection of related code (methods) and data (fields, a.k.a. instance variables).
- A *class* is a set of objects. A class definition defines what is common to that set.
- Java convention is for class names to begin Uppercase and continue camelCase.
- In file ClassName.java , the structure of a typical definition for class ClassName is:

```
public class ClassName {

    Field Definition
    Field Definition
    ...
    Field Definition

    Method Definition
    Method Definition
    ...
    Method Definition
}
```

- A *field* or *instance variable* is a variable that is an attribute of each object in the class. Each object has its own distinct variable of the given name. The variable is accessible whenever the object is accessible. A Field Definition has the following form:

  Access Specifier Type Variable Name = Initial Value ;

  - Access Specifier is usually **`public`** or **`private`**. One may get and set public variable values from any Java code in any class. When a variable is private, one may only access

the variable within the code of its class. When the access specifier is not given, default **package** access is assumed, which is slightly more restrictive than **public**.

- The Type may be a fundamental type (e.g. int, double) or an object type (e.g. any class name, such as String).
- The = Initial Value portion assigns the initial value of the variable. If this portion is not supplied, the default values for numeric types, Boolean types, and object types are 0, false, and null (meaning "no object"), respectively.

- A *method* (a.k.a. function, procedure, etc.) is a block of code "called" (i.e. caused to execute) in the context of an object, with access to the object's fields and methods. It takes input in the form of arguments evaluated and assigned to parameter variables, and returns a single output return value. When there is no output, the return type is void. The method's parameter and local variables cease to exist after the method's return. A Method Definition has the following form:

Access Specifier Return Type Method Name (Type Variable Name , … , Type Variable Name) {
    Statement
    Statement
    …
    Statement
    return Expression ;
}

  - Access Specifier is usually **public** or **private**. One may call public methods from any Java code in any class. When a method is private, one may only call the method within the code of its class. When the access specifier is not given, default **package** access is assumed, which is slightly more restrictive than **public**.
  - The Return Type may be a fundamental type (e.g. int, double) or an object type (e.g. any class name, such as String), and must match the type of the return Expression . If there is no return, the return type is "void".
  - When a method is called, the comma-separated arguments in parentheses after the method name in the call are evaluated and assigned to the formal parameters defined in comma-separated Type Variable Name pairs in the header of the method declaration. It is as though these are local variables that are initialized with the values passed as arguments in the method call.
  - When the method returns a value, that value is used in place of the original method call.

- The **static** keyword after the Access Specifier indicates that the variable/method is *associated with the class*, as opposed to being associated with a particular object of that class. Thus, one need not create an object to access/use a **static** element. Think of a **static** variable as a single variable attribute of the set of all objects of the class, and a **static** method as a single method relevant to the class of objects but not any single object in particular. Examples:

Math.PI, Math.sin(Math.PI / 2) Note that there is no creation of a Math object in order to use these.

Code Block

Any place a Statement can appear in code, you can also use a brace-enclosed *block* (i.e. sequence) of statements of the form:

```
{
    Statement
    Statement
    …
    Statement
}
```

Decisions

- **if** statement: If a Boolean (true/false) `Condition` is true, execute a `True Case Statement`:

```
if (Condition)
      True Case Statement
```

- **if-else** statement: If a `Condition` is true, execute a `True Case Statement`, else (i.e. otherwise) execute a `False Case Statement`:

```
if (Condition)
      True Case Statement
else
      False Case Statement
```

- **if-else** chain statement: If `Condition 1` is true, execute a `Case 1 Statement`, else if `Condition 2` is true, execute a `Case 2 Statement`, …, else execute a `Default Case Statement`:

```
if (Condition 1)
      Case 1 Statement
else if (Condition 2)
      Case 2 Statement
...
else
      Default Case Statement
```

- **if-else** chain statement: If `Condition 1` is true, execute a `Case 1 Statement`, else if `Condition 2` is true, execute a `Case 2 Statement`, …, else execute a `Default Case Statement`:

```
if (Condition 1)
```

```
        Case 1 Statement
else if (Condition 2)
        Case 2 Statement
...
else
        Default Case Statement
```

Loops

- **while** statement: While a Boolean (true/false) `Condition` is true, repeatedly execute a `Statement`:

  ```
  while (Condition)
        Statement
  ```

  Note: This is essentially a repeating **if** statement.

- **for** statement: After executing an `Initialization` statement, while a expression `Condition` is true, execute a `Statement` and execute a `Change` statement:

  ```
  for (Initialization; Condition; Change)
        Statement
  ```

- **do-while** statement: Repeatedly execute a `Statement` until a `Condition` is false:

  ```
  do
        Statement
  while (Condition);
  ```

  Note: This is essentially a **while** statement that always executes the `Statement` at least once.

- **for-each** statement (a.k.a enhanced for loop): For each element of a `Collection of Type`, assign the element to `Variable Name` and execute `Statement`:

  ```
  for (Type Variable Name : Collection of Type)
        Statement
  ```

- **break** statement: The statement "**break;**" will cause a loop to immediately terminate, resuming execution after the loop.
- **continue** statement: The statement "**continue;**" will cause the current loop iteration to immediately terminate, resuming execution with the next iteration.

Arrays

- arrays: An array is *list* variable, holding a predetermined number of values of a given type.
  - Example 1: `String[] a = {"This", "is", "a", "String", "array."};`

        - Here, we declare a String array (written "`String[]`") called "`a`", and initialize it to be an array of 5 given `String` objects.
    - Example 2: `int[] data = new int[10];`
        - Here, we declare "`data`" to be an integer array (written "`int[]`") of length 10.
    - The length of the array can be accessed by appending "`.length`" to the array (e.g. `a.length`, `data.length`).
    - We access each value of an array by indexing the variable name with a number (e.g. `a[0]`, `data[3]`). Java, like most other language, uses 0-based indexing. That means that the first member of the array is at index 0, and the last member of the array is an index (`array.length` – 1). Thus `a[0]` is `"This"` and `a[4]` is `"array."`.
    - When an array's values are not initialized (as with `data`), Java initializes them the same way as it does uninitialized object fields (e.g. `0` for `ints`, `false` for `booleans`, `null` for `Objects`).
- multidimensional arrays: One can create a multidimensional arrays, i.e. arrays of arrays, arrays of arrays of arrays. Each subarray need not have the same length. Examples:
    - ```
      int[][] divisors = {{}, {1}, {1, 2}, {1, 3}, {1, 2, 4}, {1, 5},
      {1, 2, 3, 6}};
      ```
    - ```
      int[][] divisors2 = new int[7][];
      divisors2[0] = new int[0];
      divisors2[1] = new int[1];
      divisors2[1][0] = 1;
      divisors2[2] = new int[2];
      …
      ```
- Arrays class: The `Arrays` class has utility functions that allow you to search, fill, sort, and create String representations of arrays.
- `ArrayList`: Often, you'll want to be able to build a list of values, but you won't know in advance what length that list should be, for such purposes, use the `ArrayList` class:
    - `ArrayList`s can only hold objects, so in order to hold fundamental types (e.g. int, double, boolean), we need to wrap them up in "wrapper classes" (e.g. Integer, Double, Boolean). Java does this for us via what is called "autoboxing". When we need to treat them again as fundamental types, Java unwraps ("autounboxes") them.
    - Example: Let's suppose you want to build and print a list of positive int divisors of an int held in variable n:
      ```
      int n = 100;
      ArrayList<Integer> divisors = new ArrayList<Integer>();
      for (int i = 1; i <= n; i++) // build a list of positive divisors of n
          if (n % i == 0)
              divisors.add(i);
      for (int divisor : divisors)
          System.out.println(divisor);
      ```
    - In the example above, we accessed the list values using the enhanced for loop ("for each" loop). We can also access items by 0-based index, so the last two lines above could have been accomplished this way:

```
for (int i = 0; i < divisors.size(); i++)
    System.out.println(divisors.get(i));
```

- Array (`a`) versus ArrayList (`al`) analogs:
  - `a.length` versus `al.size()`
  - `a[2]` versus `al.get(2)`
  - `a[3] = 5` versus `al.set(3, 5)`
- Command Line Arguments: In main method code "`public static void main(String[] args)`", `args` refers to a String array filled with command line arguments after "`java`" and the class name and separated by whitespace. Consider the following PrintArgs.java code:

```
public class PrintArgs {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.printf("args[%d]: %s\n", i, args[i]);
    }
}
```

The output printed from the console command "`java PrintArgs testing 1 2 3`" would be:

```
args[0]: testing
args[1]: 1
args[2]: 2
args[3]: 3
```