# Java Review
# Outline

- basics
- exceptions
- variables
- arrays
- `modulo` operator
- `if` statements, booleans, comparisons
- loops: `while` and `for`

# Java basics

- write a simple program, e.g. hello world

  http://www2.hawaii.edu/~esb/2021spring.ics211/HelloWorld.java.html
- how many classes does this program create?
- how many objects can you find in this program?
- how many methods are implemented by this program?
- where does a program begin execution?
- how do you declare that?
- what is the file name?
- how do you print a string?

# Java development environment: no IDE

- edit a text file using an editor
  - which editors have been used by this class?
- then **compile** using?
- then **execute** using?
  - JDK can compile, JRE (included in JDK) can execute
- compilation errors are not the same as execution errors: how do they differ?

# Java development environment: IDE

- Integrated Development Environment
- includes editor, compiler, execution engine
- Eclipse, jGrasp, NetBeans

- the Java coding standard helps other people read your program, no matter what IDE you and they use

# Java errors

- humans make mistakes: *errare humanum est*
- **compiler errors** are due to programmer lack of knowledge, to sloppiness, or to hurry
- **run-time errors** are due to programmer sloppiness, hurry, or lack of understanding
- typical run-time errors may be due to unexpected inputs, e.g. a non-numeric string where a number was expected
- avoiding runtime errors may require code to check values to make sure they fit your expectations, and have a sane response if they don’t

# Java Exceptions
(book, Appendix A.11)

- `try` block followed by one or more `catch` blocks

```
try {
  /* code which may throw exception */
}
catch (exceptionClass1 variable1) {
  /* code which handles exception of type exceptionClass1 */
}
catch (exceptionClass2 variable2) {
  /* code which handles exception of type exceptionClass2 */
}
finally {
  /* code executed after try or catch */
}
```

# Java Exceptions II

- example (from the textbook, p. 608)

```
try {
// statements that perform file-processing operations
}
catch (IOException ex) {
  ex.printStackTrace();  // display stack trace
  System.exit(1);        // exit with an error indication
}
```

- an uncaught exception is caught by the runtime system: this is a run-time error (but not all run-time errors result in exceptions)
- an example showing try/catch and command-line arguments

  http://www2.hawaii.edu/~esb/2021spring.ics211/Substring.java.html
- another example program using try/catch and GUI I/O

  http://www2.hawaii.edu/~esb/2021spring.ics211/ChangeDue.java.html

# Java variables

- local variables only have scope from their declaration to the end of the block
- parameters have scope within the method
- smaller declarations override larger declarations, but: this is bad programming practice, since a reader may not know which of multiple equally-named variables the programmer is using, e.g.

```
private void countUpTo (int count)
{
  count = 0;
  while (count < 1000) {
    int count = 15;
    count = count + 3;
  }
}
```

- when there is more than one such variable, this code will refer to the innermost count variable
  - more recent versions of Java actually make this an error

# Java arrays

- arrays are objects
- that contain other objects, the array elements
  - or arrays can contain elements of one of the basic types
- all elements of an array have the same type
- each element is identified by an integer value in (0...array.size-1), and the array index or subscript
- like any variable, arrays must be declared
- unlike integers or booleans, but like any object, arrays must be initialized, that is, allocated or instantiated, e.g.

```
String[] a = new String [15];
boolean[] bits = new boolean [99];
```

- the first expression creates an array of size 15 with indices 0..14
- the expression "a [7]" has what type?
- see the ChangeDue example

# array initialization

- an array that is not initialized to an allocated value has the special value `null`, which cannot be indexed
- initializing the array (the container) is different from initializing each of the elements: the array must be initialized first, then the elements
- initializing array elements one by one can be as follows

```
coinNames[4] = "half";
```

- This could be done within a loop, e.g.

```
a[i] = i*7;
```

- or all can be initialized and allocated at once, e.g.

```
double[] x = {1.0, 2.71, 3.14, 4.0};
```

# array operations

- the length of an array is one more than the highest usable index

```
int i = 0;
while (i < x.length) {
   x [i] += 1.0;
}
```

- Arrays.copyOf can be used to copy elements from one existing array to another existing array, and also to extend or shorten an array:

```
T[] Arrays.copyOf(T[] original, int newLen);
```

- if the array is extended, new elements are assigned the default value: null, 0, or false

# The modulo (%) operator

- integer division gives two results: a quotient and a remainder
- for example 11/4 = 2 with a remainder of 3
- the quotient is given by /

```
int quotient = 11 / 4;    // 2
```

- the remainder is given by %, known as modulo

```
int remainder = 11 % 4;  // 3
```

- this code computes a random bit, 0 or 1:

```
Random myRandomGenerator = new Random();
int random = myRandomGenerator.nextInt() % 2;
```

- modulo will be used throughout this class
  - because we can turn any positive integer into a valid array index by taking it % array.length
  - e.g., given

    `String[] a = new String [15];`

    `n % a.length` is a valid index, i.e. can access `a[n % a.length];`
- if the divisor is negative, the modulo/remainder will be negative

# example using modulo

- suppose I want to use an array in a circular fashion
- so that the last location in the array is followed by the first location in the array
- For example, suppose the array had values for traffic light signals:

```
String [] lights = { "green", "yellow", "red" };
int lightIsNow = 0;
```

- then, to cycle through the lights, all we need is:

```
lightIsNow = (lightIsNow + 1) % lights.length;
```

- if light is red (`lightIsNow` has the value 2), the next light will be green (`lightIsNow` has the value 0)

# booleans and boolean expressions

- variables of type boolean have either the value true or the value false
- expressions of type boolean have either the value true or the value false
- for example, x < y is either true or false
- arithmetic values or expressions can be used in comparisons:
- < <= == != >= >
- for objects, == compares the addresses -- are these two the same object -- whereas the equals method may perform a different comparison (and is usually the one to use)
- boolean expressions or values can be combined using the logical operators:
- && || !
- boolean expressions and values are used to determine which code gets executed when in conditional statements and loops

# conditional statements

```
boolean b = false;
if (x > y) {
  a = 3.0;
} else if (! b) {
  a = 2.2;
} else {
  x = 99;
}
```

- the part after the first if is executed if the first condition is true
- otherwise the part after the first else is executed
- in-class exercise: determine the output of this program: http://www2.hawaii.edu/~esb/2013fall.ics211/Equals.java.html

# loops

- These two sets of statements are equivalent:

```
int x = 99;
while(x > 0) {
   removeBottleOfBeerFromWall();
   x = x - 1;
}
```

- and

```
for (int x = 99; x > 0; x--) {
        removeBottleOfBeerFromWall();
}
```

# loops, continued

- a `while` loop is good when something needs to be done zero or more times:

```
while(x > 0) {
  x -= y;
  y = y * 2;
}
```

- `do .. while` can be used to execute statements one or more times:

```
do {
  x -= y;
  y = y * 2;
} while(x > 0);
```

  a `for` loop is particularly useful with loop variables:

```
for (int i = 0; i < a.length; i++) {
  a [i] *= 2;
}
```

- what does this program do?

  http://www2.hawaii.edu/~esb/2021spring.ics211/MysteryProgram1.java.html

# loops, continued

- a `while` loop is good when something needs to be done zero or more times:

```
while(x > 0) {
  x -= y;
  y = y * 2;
}
```

- `do .. while` can be used to execute statements one or more times:

```
do {
  x -= y;
  y = y * 2;
} while(x > 0);
```

a `for` loop is particularly useful with loop variables:

```
for (int i = 0; i < a.length; i++) {
  a [i] *= 2;
}
```

# summary

- Java primitives you should already be familiar with
- anything you are not familiar with, become familiar with, by using the course resources: the TA in lab, ATA, LA, the instructor, the course web page, the textbook (Appendix A), and the reference (The Java Language Specification).
- Be sure you know the material in Appendix A of the textbook. Please skim it and make sure you are comfortable with all the subjects in the appendix.