CSC 102 Lecture Notes Week 2
Introduction to Incremental Development and Systematic Testing
More Java Basics

Revised 12 April

I. **Relevant reading.**

A. Horstmann chapters 1 - 6 (continued from last week).

B. Writeups for Lab 3, Lab 4, and Program 1.

C. Various cited material in the writeups.

II. **The basic idea of incremental development and systematic testing.**

A. The idea of *incremental development* is to develop a program in a step-by-step process.
   1. You can start by laying out an overall design of the program
      a. first determine the classes you'll need in the program
      b. then determine the methods you'll need within the classes.
   2. Then you can develop the methods one at a time, starting with the simplest and most basic methods first, progressing to the more complicated methods one-by-one.

B. The idea of *systematic testing* is to make sure that the classes get tested thoroughly and completely.
   1. An effective way to do this is to test each method as it's being developed.
   2. Testing a method entails calling it a bunch of times to make sure it does what it's supposed to do.
   3. The definition of "what it's supposed to do" comes from the program specification.

III. **Typical design of Java testing programs.**

A. It's common Java practice to organize classes in pairs -- one class to be tested, another class to do the testing.

B. For example, for the first program we have the class `Fraction` to be tested, and the class `FractionTest` that will do the testing.

C. It is a very common Java naming convention to have the name of the testing class be the same as the name of the class being tested, with the suffix "`Test`" added to the name.

IV. **A basic plan for testing a class.**

A. The overall goal is to test all of the public methods.

B. To get this done, the following is a typical order of method implementation and testing:
   1. Implement any necessary access methods in the class being tested.
      a. Such methods are necessary to provide access to private data fields, so the testing methods can check that results are correct.
      b. For example in the case of the `Fraction` class, implement `getNumerator` and `getDemonimator` first.
   2. Test the class constructors first.
      a. This makes sense since you need to construct objects before you can test the methods in those objects.
      b. In the case the `Fraction` class, the first tests you'll do are for the three `Faction` constructors -- `testDefaultConstructor()`, `testNumeratorConstructor()`, `testNumeratorDenominatorConstructor()`.
   3. Next you can test methods that work with values without modifying them.
      a. It's reasonable to test these methods next because in many cases they may get used by other methods in the class.
      b. The idea is to test methods first that other methods may rely on.

   c. In the case of the `Fraction` class, you can write `testToString` and `testEquals` after you have fully tested all three of the constructors.

4. Finally, test methods that perform computations on class data.
   a. It makes some sense to test these methods later, since they are often more complicated, and my rely on previously tested methods.
   b. The idea here is develop and test incrementally from simple methods to more difficult ones.
   c. In the case of the `Fraction` class, testing of the computational methods is done by `testAdd()`, `testSub()`, `testMul()`, and `testDiv()`.

V. **What's in a testing method?**

A. Testing a method is done in a sequence of *test cases*.

B. Each *test case* involves:
   1. Selecting input(s) for the method to be tested.
   2. Determining what you expect the method to output given the selected inputs.
   3. Calling the method with the inputs to see if it actually outputs what you expect.

C. The are a number of different ways to implement this kind of testing; one common approach goes like this:
   1. Set up the necessary inputs, including constructing objects if necessary.
   2. Call the method.
   3. Use an `if` statement to compare the actual output with expected output and print an error message if they don't agree.

D. Here's a concrete example of three test cases for `testNumeratorDenominatorConstructor()`, in a very simple version of the `TestFraction` class:

```
/****
 * Class SimpleFractionTest is a very small example illustrating what your
 * FractionTest class can look like for Programming Assignment 1.
 */
public class SimpleFractionTest {

    /**
     * Call the test method for testNumeratorDenominator.  In the complete
     * FractionTest you’re writing, this main method calls all of the faction
     * test methods.
     */
    public static void main(String[] args) {
        testNumeratorDenominatorConstructor();
    }

    /**
     * Test the full initializing constructor of the Fraction class with three
     * sample test cases.  In the full version of this test method you’ll need
     * some additional test cases.  Use the guidelines in Lecture Notes 2 to
     * help figure out what the additional test case should be.
     */
    private static void testNumeratorDenominatorConstructor() {

        Fraction f;   // value produced by the constructor
        int n;        // convenience varible for the numerator value
        int d = 0;    // convenience varible for the denominator value

        // Test Case 1: check the boundary case of a zero numerator "0/1".
        f = new Fraction(0,1);
        if ((n = f.getNumerator()) != 0 || (d = f.getDenominator()) != 1) {
            System.out.println("Got " + n + "/" + d + ", expected 0/1");
        }

        // Test Case 2: check a simple case the doesn’t need reduction.
```

```
            f = new Fraction(1,2);
            if ((n = f.getNumerator()) != 1 || (d = f.getDenominator()) != 2) {
                System.out.println("Got " + n + "/" + d + ", expected 1/2");
            }

            // Test Case 3: check a case that needs some reduction.
            f = new Fraction(4,8);
            if ((n = f.getNumerator()) != 1 || (d = f.getDenominator()) != 2) {
                System.out.println("Got " + n + "/" + d + ", expected 1/2");
            }

        }

    }
```

E. And if the following simplification of this code isn't screaming at you, it should be:

```
/****
 * Class SimpleFractionTest is a very small example illustrating what your
 * FractionTest class can look like for Programming Assignment 1.
 */
public class SimpleFractionTest {

    /**
     * Call the test method for testNumeratorDenominator.  In the complete
     * FractionTest you’re writing, this main method calls all of the faction
     * test methods.
     */
    public static void main(String[] args) {
        testNumeratorDenominatorConstructor();
    }

    /**
     * Test the full initializing constructor of the Fraction class with three
     * sample test cases.  In the full version of this test method you’ll need
     * some additional test cases.  Use the guidelines in Lecture Notes 2 to
     * help figure out what the additional test case should be.
     */
    private static void testNumeratorDenominatorConstructor() {

        Fraction f;   // value produced by the constructor

        // Test Case 1: check the boundary case of a zero numerator "0/1".
        test(0, 1, 0, 1);

        // Test Case 2: check a simple case the doesn’t need reduction.
        test(1, 2, 1, 2);

        // Test Case 3: check a case that needs some reduction.
        test(4, 8, 1, 2);
    }

    /**
     * Output an error if the given Fraction f does not have the given
     * values for nExpected and dExpected for its numerator and denominator.
     */
    private static void test(int nIn, int dIn, int nExpected, int dExpected) {
        int n;        // convenience variable for the numerator value
        int d = 0;    // convenience variable for the denominator value
        Fraction f = new Fraction(nIn, dIn);

        if ((n = f.getNumerator()) != nExpected ||
                (d = f.getDenominator()) != dExpected) {
```

```
                System.out.println("Got " + n + "/" + d +
                    " expected " + nExpected + "/" + dExpected);
            }
        }
    }
```

VI. **Some initial thoughts on choosing good inputs.**

A. In the description above of systematic testing, we said that it needs to be *thorough* and *complete*.

B. One of the key aspects of this is selecting good inputs for test cases.

C. It turns out that this is a really large subject in computer science that we’ll only just start looking at in 102.

D. For starters, here are a few well-accepted guidelines for selecting good test inputs:

1. Test ranges of input values.
   a. "Smallest" possible value.
   b. "Largest" possible value.
   c. Values in between.
   d. I.e., test at the *boundaries* and selected points between.
2. For mid-range values,
   a. Choose typical or normal values for inputs, based on the program specification.
   b. Choose *representative* values, to avoid redundancy (more on this coming weeks).
3. As you test methods:
   a. Test different combinations of inputs.
   b. Test with inputs that cover all parts of your code.
   c. Test with inputs that exercise the "tricky parts" of your code (there’s actually a science to this, but it sometimes feels like an art).
   d. Test with inputs that cause exceptions.

That’s it for our the introduction to testing.
We’ll have plenty more to say about it as the quarter goes on.
In the meantime for these notes, we’ll resume our discussion on
the basics of object-oriented programming in Java.

VII. Data values in Java.

A. There are two kinds of data in a Java program -- *primitive data* and *class objects*.

B. For the programs we’ll write in 102, the primitive types we’ll use are `int`, `double`, `boolean`, and `char`.

1. There are additional primitive types named `float`, `byte`, `short`, and `long`.
2. Table 1 in Chapter 4 (page 129) describes all of these primitive types.

C. Anything other than a primitive data value is represented as an *object*.

1. An object is the value of a type defined as a class, i.e., a type that is not one of the eight primitives.
2. Array values are also non-primitive, and hence array values are objects.

D. Objects are created with the `new` operator.

1. Specifically, whenever `new` is executed, a brand new object is created.
2. As special cases, string and array objects can be created without using `new`.
   a. Literal string objects can be created using double quotes, in the normal way.
   b. Array objects can be created using curly braces, as in C.
   c. We’ll look more closely at string and array literals later in these notes.

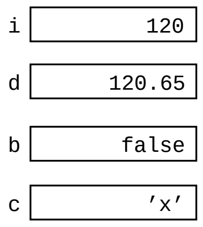
VIII. **Where data are stored.**

A. Primitive data values are stored directly within the variable or parameter to which they are assigned.

1. For example, consider the following primitive declarations:

```
int i = 120;
double d = 120.65;
boolean b = false;
char c = 'x'
```

2. These are pictured in memory like this:

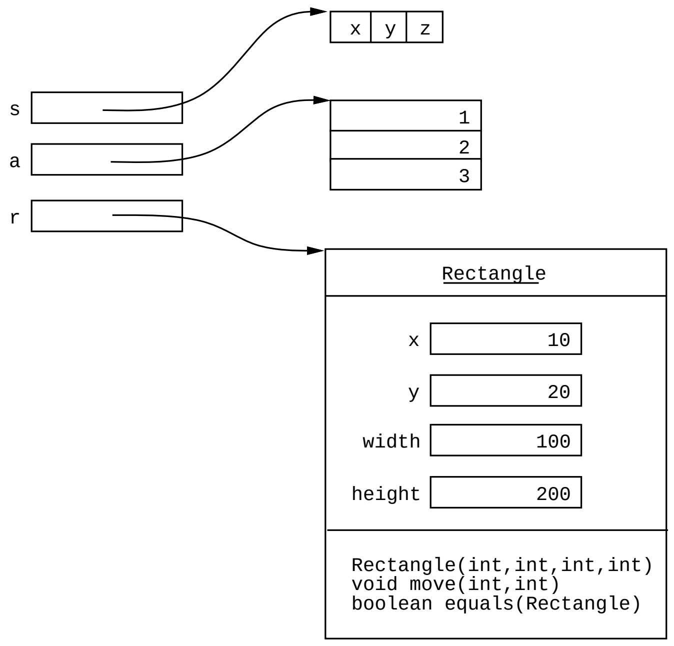


B. Object data are stored as *references* within the variable or parameter to which they are assigned.

1. For example, consider these non-primitive declarations:

```
String s = "xyz";
int[] a = {1, 2, 3};
Rectangle r = new Rectangle(10, 20, 100, 200);
```

2. These are pictured in memory like this:



IX. **How objects are created.**

A. A class object is created using the `new` operator, followed immediately by a call to a class constructor.

B. A constructor is a special form of method that is called whenever a class object is created.

C. Syntactically, a constructor is declared like a method, using the same name as the class, without return value.

D. The "return" of a constructor is a class object, which contains its data fields and methods.

E. I.e., a constructor returns the `this` object for a class.

F. Constructors can be *overloaded*, there can be more than one definition of the same constructor, as long as the input parameters are different.

1. A typical case is to have two overloads of a constructor -- one with no parameters, and another with one parameter for each data field.
2. A zero-argument constructor is often called a *default* constructor
   a. In fact, if a class has not explicitly declared any constructor at all, Java defines a zero-argument constructor ***by default***, i.e., Java automatically declares a default constructor.
   b. Even though Java will automatically declare a zero-argument default constructor, it is considered good programming practice always to declare one explicitly, so it can be documented, and so it can perform any necessary default data initializations.
   c. Furthermore, if you define any constructor(s) at all, Java doesn't give you the default constructor for free any more, so you'll need to define it yourself. (*Think about this a bit.*)
3. A constructor with one argument for each data field is called an *initializing* constructor.
   a. A good example is the constructor for the `Rectangle`:

```
Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

   b. We'll say more about this in upcoming discussions.

X. **Class member visibility.**

A. Java provides three levels of visibility for methods and data fields within a class:
   1. ***public*** -- visible in any other class
   2. ***protected*** -- visible in a restricted subset of other classes
   3. ***private*** -- not visible in any other class

B. For the next few weeks in 102, we will follow the following visibility rules:
   1. The following are declared `public`:
      a. all classes, i.e., the public modifier goes at the beginning of all class definitions
      b. all constructors
      c. all methods in the API that other classes use
   2. The following are declared `private`:
      a. all data fields, i.e., instance variables
      b. all methods that are used in the class, but not provided in the API
   3. For now, we will not use the `protected` form of visibility.
   4. Note that methods and data fields without an explicit `public` or `private` declaration are not allowed; this form of visibility is called "package protected", which is not something we want right now.
   5. Note also that method local variables are not declared `public` or `private`, since they are only visible inside the method in which they are declared.

XI. **Accessor and mutator methods.**

A. Given our requirement that data fields are all private, accessing and changing class data fields must be provided by public *accessor* and *mutator* methods.
   1. These are introduced in Section 2.7 of the book.
   2. For example, simple accessor and mutator methods for the `Rectangle` class discussed in notes 1 would look like this:

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;
    //
    // Simple accessor methods are typically called "getters"
    //

    /**
     * Return the x coordinate value.
     */
    public int getX() {
        return x;
    }

    /**
     * Return the y coordinate value.
     */
    public int getY() {
        return y;
    }

    /**
     * Return the width.
     */
    public int getWidth() {
        return width;
    }

    /**
     * Return the height.
     */
    public int getHeight() {
        return height;
    }


    //
    // Simple mutator methods are typically called "setters".
    //

    /**
     * Set the x coordinate to the given int value.
     */
    public void setX(int x) {
        this.x = x;
    }

    /**
     * Set the y coordinate to the given int value.
     */
    public void setY(int y) {
        this.y = y;
    }

    /**
     * Set the width to the given int value.
     */
    public void setWidth(int width) {
        this.width = width;
    }

    /**
     * Set the height to the given int value.
     */
    public void setHeight(int height) {
        this.height = height;
    }

    // ... other methods of the Rectangle class, as shown in ../Rectangle.java

}
```

B. When looking at one-line getter and setter methods like this, you might say "Why not just make the data fields public, since this these methods are making data *effectively* public anyway?"

C. We will address this question a bit later in the quarter, when we discuss ***data abstraction***, and explain why access and mutation through methods is a better idea than public data.

XII. **How data are compared for equality.**

A. Comparing values for equality is very important topic in any programming language, and particularly so in an object-oriented language.

B. Primitive data values are compared with the "`==`" operator, just as they are in C.

1. For example, given the primitive variable definitions above,

```
int i = 120;
double d = 120.65;
boolean b = false;
char c = ’x’;
```

the following are some sample comparisons:

```
i == 120;       // true
i == 120.65;    // false
i == d;         // false
i == c;         // true
d == c;         // false
c == b;         // compilation error: incompatible types
i == b;         // compilation error: incompatible types
d == b;         // compilation error: incompatible types
```

2. Note that ints, doubles, and chars are type compatible for comparison purposes, but not none of these three types is compatible with boolean.

3. Also, have a close look at the true/false values of these primitive equality expressions (in the // comments) and convince yourself that you understand what’s going on.

C. Non-primitive object values can be compared with `==`, or with the `.equals` method.

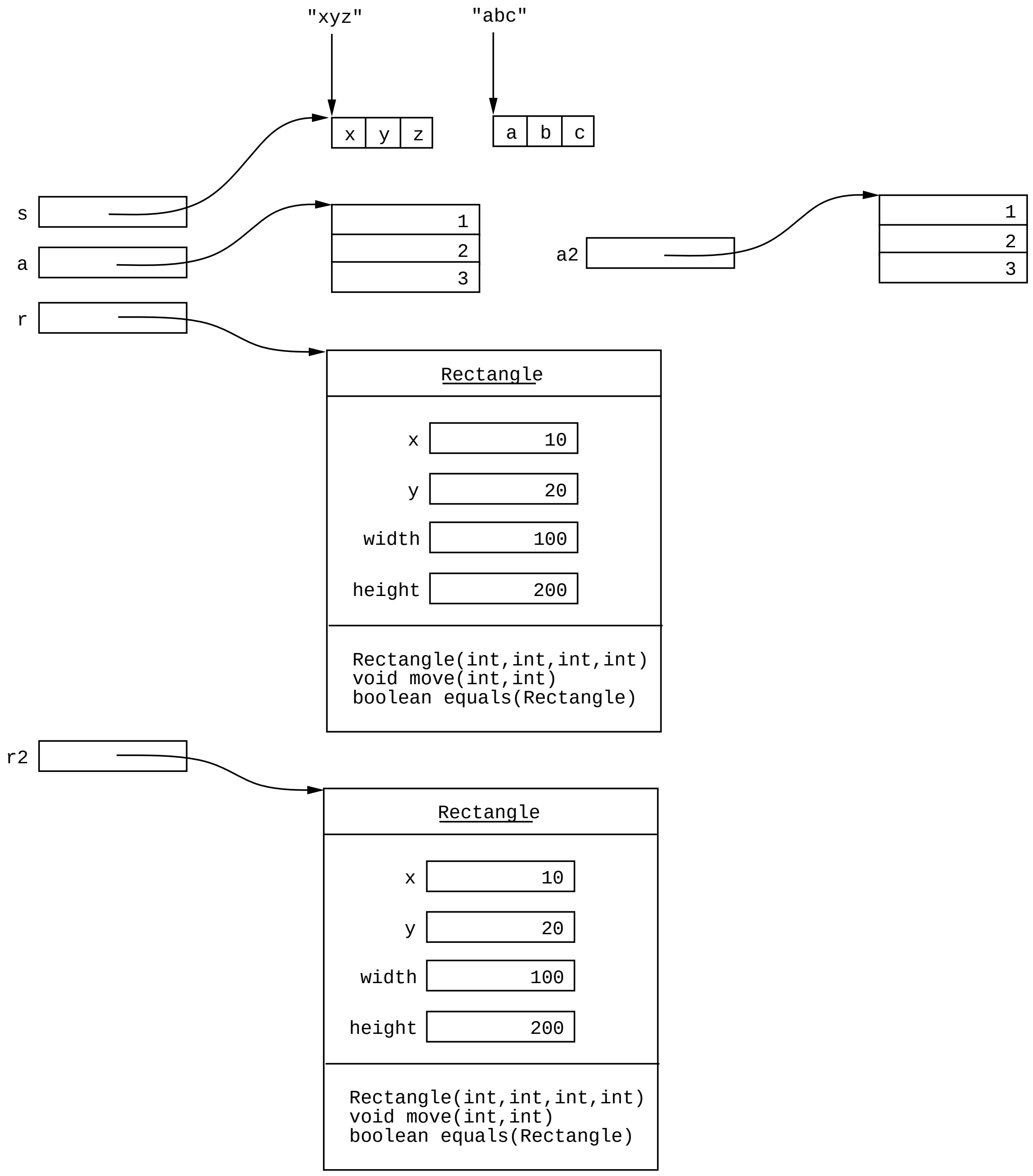
1. For example, consider the non-primitive variable definitions above for String `s`, array `a`, and Rectangle `r`

```
String s = "xyz";
int[] a = {1, 2, 3};
Rectangle r = new Rectangle(10, 20, 100, 200);
```

2. The following are some sample comparisons for these variables:

```
s == "xyz";             // true
s == "abc";             // false

int[] a2 = {1, 2, 3};
a == a2;                // false
a.equals(a2);           // false ( .equals on arrays defaults to == )

Rectangle r2 = new Rectangle(10, 20, 100, 200);
r == r2;                // false
r.equals(r2);           // true ( this works given Rectangle.equals )
```

3. The following is a picture of memory that helps explain what’s going on in the preceding example.



D. When `==` is used to compare objects, it compares *the object references*, NOT contents of the objects; this is ***shallow*** comparison.

1. What it means to compare two references is to compare the locations to which they refer, not the contents at those locations.
2. Hence, two references are `==` if they refer to the same exact object in memory.
3. Two references are not `==` if they refer to different objects, even if the contents of the objects is the same.
4. These are things you should have observed in Lab 3.

E. When `.equals` is used to compare objects, it is up to the implementor of the `equals` method to define exactly what equality means.

1. For the built-in Java String class, `equals` is defined as characterwise equality; this is an example of ***deep*** equality
2. For the `Rectangle` class defined in the last notes, `equals` is defined as a comparison of all four

`Rectangle` data fields; this is also an example of ***deep*** equality.

F. In the preceding examples, the expression

```
r == r2
```

is false, because the two variables refer to different objects.

G. In contrast, the expression

```
r.equals(r2)
```

is true, because the `Rectangle` class implements the equals method to compare two rectangles component-wise, i.e., like this:

```
boolean equals(Rectangle r) {
    return x == r.x &&
           y == r.y &&
           width == r.width &&
           height == r.height;
}
```

H. It is important to note that the `equals` method must be explicitly implemented in a class in order for it to behave differently from ==.

1. By default, the implementation of `equals` uses the == operator ***on references***.
2. Hence, if a class does not define its own version of `equals`, the == operator and the `equals` method behave exactly the same for values of that class.
3. These are also things you should have noticed in Lab 3.
4. We'll see more examples of exactly what this means as the quarter progresses

XIII. **Details of Java String data and string comparison.**

A. Java strings are defined by the library `String` class.

B. This `String` class provides a useful set of methods for string manipulation, including indexing, concatenation, and conversions.

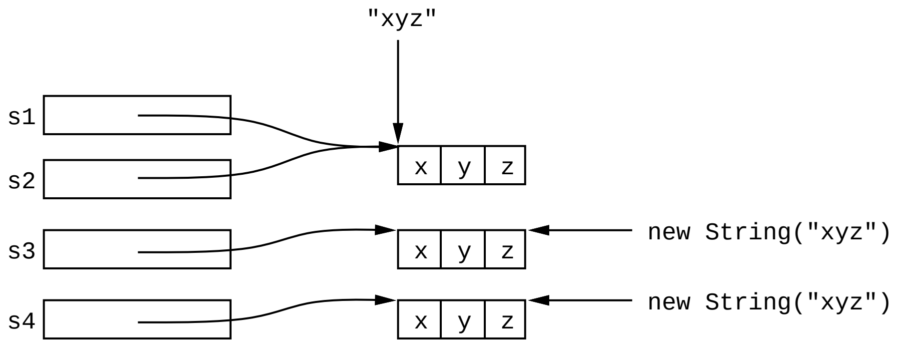
C. Section 4.5 of the book is a good summary.

D. The Java library documentation for the `String` class has the complete specification.

E. The following example program illustrates some further details of comparing strings for equality.

```
/****
 *
 * Examples of how equality works for literal string values.  See Lecture Notes
 * 2 for some explanatory discussion.
 *
 */
public class StringEquality {

    public static void main(String[] args) {

        String s1 = "xyz";
        String s2 = "xyz";
        String s3 = new String("xyz");
        String s4 = new String("xyz");

        System.out.println(s1 == s2);           // true
        System.out.println(s1 == s3);           // false
        System.out.println(s3 == s4);           // false

        System.out.println(s1.equals(s2));      // true
        System.out.println(s1.equals(s3));      // true
        System.out.println(s3.equals(s4));      // true

    }

}
```

1. A double quoted string literal is always the same object in a Java program, which explains why `s1 == s2` is true.
2. As noted above, the `String` class implements `equals` as elementwise comparison of characters, which explains why all three calls to `String.equals` return true.
3. Here's a picture of how memory is laid out for this program.



XIV. **Some initial discussion of Java arrays and array comparison.**

A. Java arrays are general-purpose structures that can contain elements of any type, but all elements of any array are the same type.

B. There is no built-in class named "`Array`" that is comparable to the built-in `String` class.

1. There is a utility named `java.util.Arrays`, more about which we'll say in next week's notes.
2. But note carefully that `Arrays` is not a type; again, more about this coming soon.

C. There is also a subtle difference between `String` and array literals

1. As noted earlier, a double-quoted string literal always refers to the same object, however many times that literal appears in a program.
2. In contrast, a curly-braced array literal creates a new array object every time it is used.

D. Another consequence of there not being a type-defining `Array` class is that there is no built-in `equals` method for all arrays.

1. This is the reason `a.equals(a2)` is false in the preceding example.
2. Hence, `equals` for any two arrays always defaults to `==`, that is, reference comparison, ***not*** elementwise comparison.
3. To get elementwise equality for arrays, you must write your own for loop, as illustrated in an example next week.
4. There is also a built-in `Arrays` class in Java that has an equals method; we will discuss this class a bit later in the quarter.
5. Don't worry if you're confused and/or frustrated by these details of arrays; so are just about every other Java programmer!
6. A good deal of the frustration is overcome by using the `ArrayList` class, about which will say much more in upcoming lectures.

XV. **Miscellaneous Java topics covered in the book, but not fully discussed in** lecture.

A. Terminology synonyms:

1. *method* = *function*

2. *reference = pointer*
3. *keyword = reserved word*
4. *data field = instance variable*

B. Defining constants.

1. Constants are defined in Java as `final` data fields.
2. E.g.,

```
private final int MAX_TRANSACTIONS = 100;
```

C. String concatenation, particularly for printing via `toString` methods.

1. The '`+`' on strings performs string concatenation.
2. E.g., `"x" + "y" + "z"` results in the string `"xyz"`.
3. Conveniently, the '`+`' operator automatically converts non-strings to strings for concatenation purposes.
4. For example, if the values of integer variables `i` and `j` are `10` and `20` respectively, then the expression

   ```
   "The value of i and j are " + i " and " + j "."
   ```

   is the string `"The value of i and j are 10 and 20."`
5. These are things related to some of the questions in Lab 3.

D. The bare bones basics of exception handling.

1. In the labs and programs so far, we've touched on the idea of exception handling.
2. Using exceptions in Java has two parts:
   a. When a method detects an error, it can ***`throw`*** an exception.
   b. When someone calling that method wants to handle the exception, it uses a ***`try-catch`*** statement
3. In program 1, you need to use `throw` an `IllegalArgumentException` if someone tries to construct a `Fraction` with a non-positive denominator.
4. To *handle* an exception, for example in a testing program, you use the following form of *try-catch* statement:

   ```
   try {
       . . .
       method call that could throw an exception
       . . .
   catch (ExceptionName e) {
       . . .
       code that handles the exception, e.g., by printing an error message
       . . .
   }
   ```

   a. We don't get to the `try-catch` exception handling statement until chapter 11 of the book.
   b. In your version of `FractionTest.java`, you can use the following form of code for test cases where the `Fraction` constructor throws an `IllegalArgumentException`:

   ```
   try {
       // Test that exception is thrown with a zero denominator value.
       new Fraction(1, 0);
   }
   catch (IllegalArgumentException e) {
       System.out.println("IllegalArgumentException thrown as expected");
   }
   ```

E. **Introduction to program debugging.**

A. Chapter 6 of the book touches on the subject.

B. The jGrasp and Eclipse environments provide very good debugging tools.

C. In an upcoming lecture and/or lab, we'll run a demo of debugging a very simple infinite loop program, in `102/examples/InfiniteLoop.java`