Using Classes and Objects

- Objectives - when we have completed this set of notes, you should be familiar with:
  - reference types and object creation
  - the String class
  - packages and the import declaration
  - the Random class
  - the Math class
  - formatting output: NumberFormat and DecimalFormat
  - wrapper classes

Review: Primitive Types

- Recall that a variable can be used to **store** a primitive type:
  - `int number;`
    - Declares a variable **`number`** can hold a 32 bit integer
  - `number = 67;`
    - the variable number now holds a value of 67
- Recall that Java has 8 primitive types:

  byte, short, int, long - - integer types
  float, double - - floating point types
  char - - holds a single character (e.g., `'A'`,`'a'`,`'$'`)
  boolean - - values of `true`, `false`
- All other types are object (or reference) types

Objects: The Basics

- Objects are defined by classes; the type for an object is the class rather than a primitive type
  - Variables for objects are be *declared* using the class name; consider a variable for a String object

```
String title;
```

  - Objects are created with the ***new*** operator; and a variable can then be assigned to reference the object:

```
title = new String("Using Classes");
```

  - Or both *declared* and *initialized* with a *new* object:

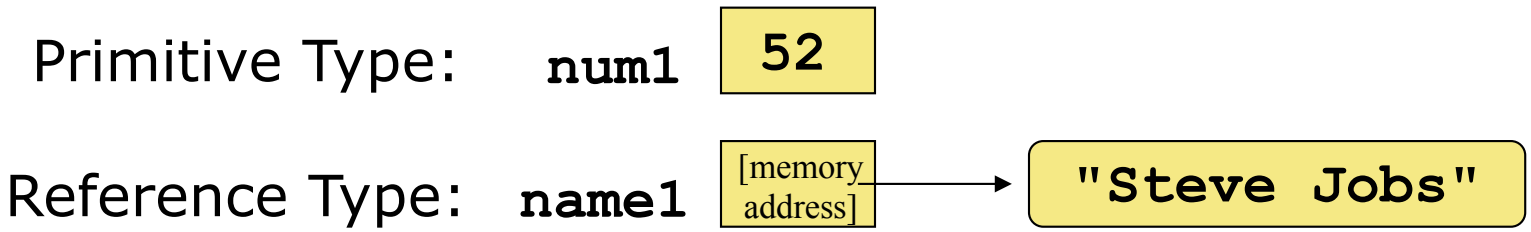
```
String team = new String("Red Sox");
```

  - The String is used so often that Java allows:

```
String location = "Shelby Center";
```

Creating Objects

- An object variable is a *reference variable*; it doesn't hold the object itself but rather the memory location where the object is stored
  - If primitive types are 'suitcases' (memory locations) that store contents, then reference variables are 'suitcases' (memory locations) that contain an address that 'points' to the location of the contents.
- Represented graphically …



Creating Objects

- Declaration does not create an object.
  - Declares that you "plan" to assign an object of this type to the variable; i.e., that the variable can reference an object of the declared type

```
String title;
```

  - Any reference type can be set to `null` to indicate that no object has been created, which allows the program to check for the existence of the object.

```
title = null; // not the same as title = "";
if (title == null) {
   System.out.println("No title set!");
}
```

Creating Objects

- The **new** operator is used to create an object

```
title = new String("Using Classes");
```

Calls a *constructor* in the String class, which initializes the new String object based on the value passed in

- Creating an object is called *instantiation*
  - creates an instance of the class
- An object is an *instance* of a particular class

```
Scanner myScan = new Scanner(System.in);
```

Invoking Methods

- Objects (unlike primitives) can have methods
  - Provide functionality - - the nextInt() method in the Scanner class reads user input as an int
  - invoked using the *dot operator* - `myScan.nextInt()`
  - A method may *return* a value:

```
int count = title.length();

System.out.println("Length is " + title.length());
```

  - A method may accept *parameters* (or args) as input:

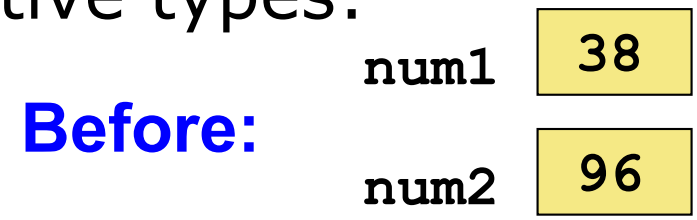
```
myScan.useDelimiter(",");
```

  - Or have both a return value and parameters:

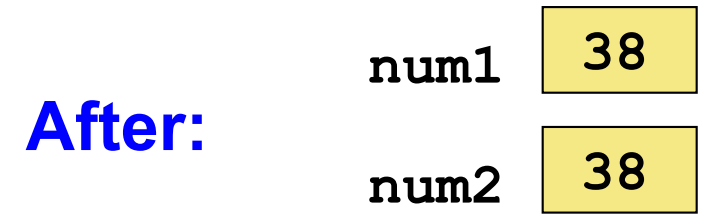
```
char singleLetter = title.charAt(2);
```

Assignment Revisited

- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:




```
num2 = num1;
```

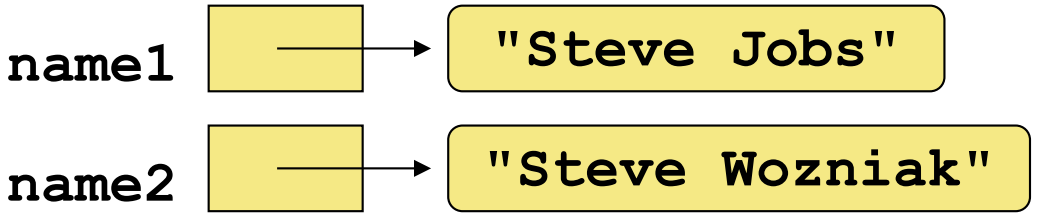


num1 and num2 both hold the same value in different memory locations

Reference Assignment

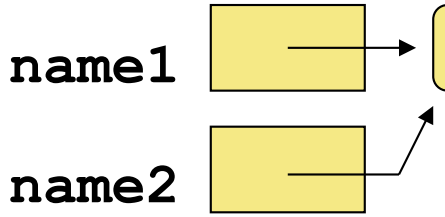
- For object references, assignment copies the address:

Before:




```
name2 = name1;
```

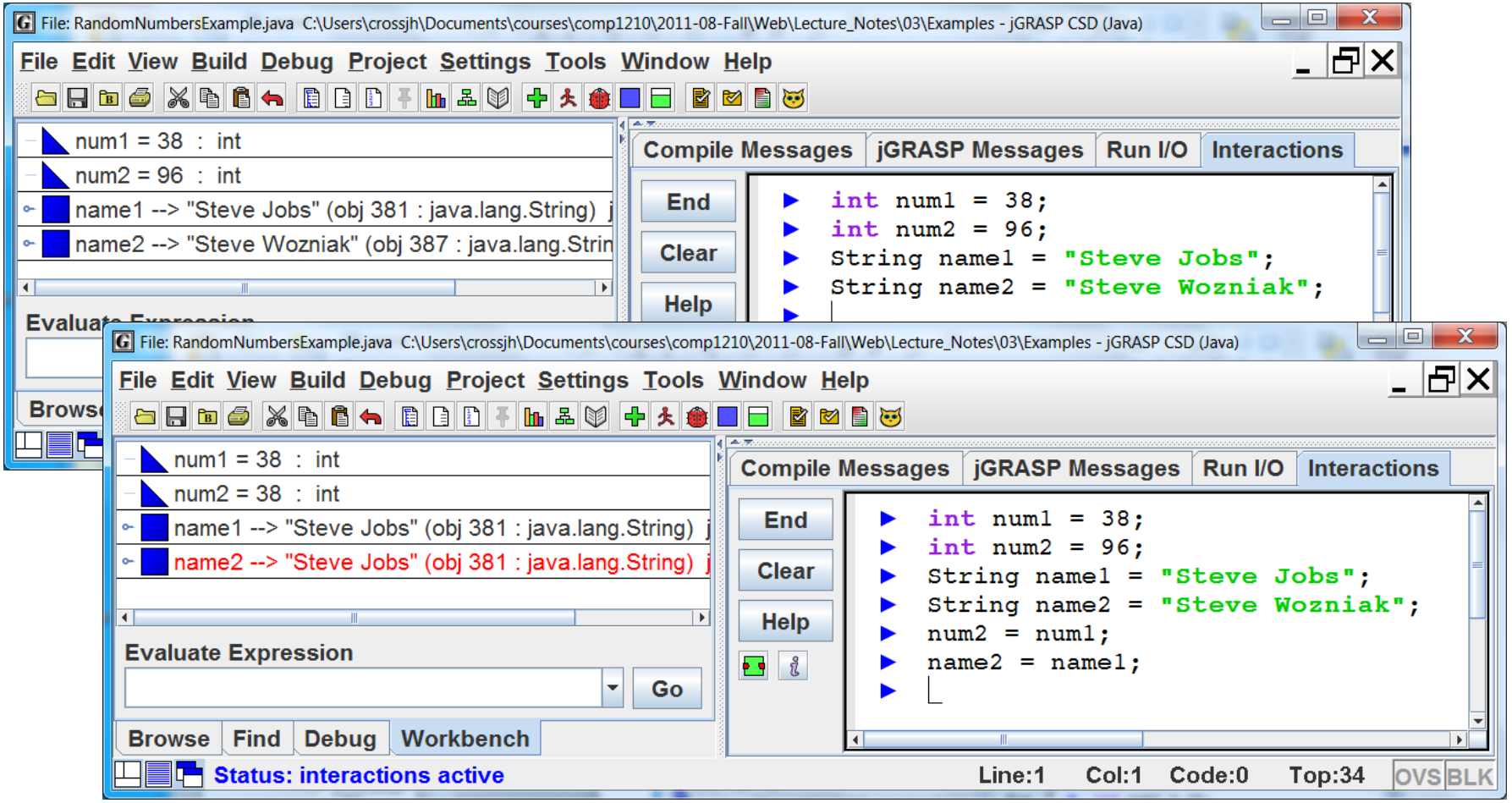
After:



name1 and name2 now both point to the same object in memory

Primitive and Reference Types - Notation in jGRASP

- Workbench and Debug tabs show difference



Aliases

- Two or more references that refer to the same object are called *aliases* of each other

```
Scanner scan1 = new Scanner(System.in);

Scanner scan2 = scan1;
```

- If you change an object using one reference, it's changed for the other reference too.

```
scan2.useDelimiter(",");
```

  scan1 will now use the same delimiter as scan2 since they reference the same Scanner object

Garbage Collection

- When an object no longer has any references to it (i.e, no variables point to it), it can't be accessed
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- Languages such as C and C++ require the programmer to perform garbage collection
  - allocation and deallocation of memory

Q1

The String Class

- String object creation (instantiation) has two forms: (1) the `new` operator and (2) the String literal (assume `title` was declared to be type `String`)

```
title = new String("Using Classes");

title = "Using Classes";
```

- Each string literal (enclosed in double quotes) represents a `String` object
- For most reference types, the `new` operator is used to call the constructor for object creation.

StringExample1.java

The String Class

- String objects are *immutable*
  - Cannot be changed in memory once created
- Ex: the replace() method returns a whole new String object (the target String is unchanged)

```
String title2 = title.replace("s", "S");
```

- The following may appear to replace all characters s with S, but it effectively does nothing

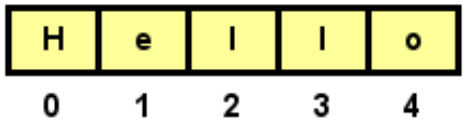
```
title.replace("s", "S");
```

  - You probably meant to do this:

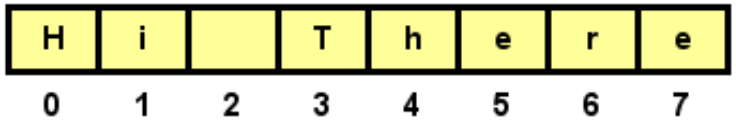
```
title = title.replace("s", "S");
```

String Indexes

- Characters are indexed starting at 0
  - In the string `"Hello"`, the character `'H'` is at index 0 and the `'o'` is at index 4



  - `"Hi There"` (spaces are characters too!)

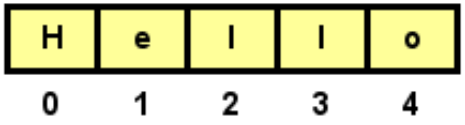


- You can get a particular character from a String using the charAt method and a substring of a String using the substring method with the indexes of the String

StringExample2.java

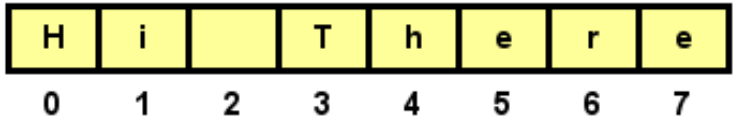
charAt and substring

- Here's the charAt method used to find the first character of a String




```
String myString = "Hello";
char myChar = myString.charAt(0);
```

- The substring method used to find the first and last half of a String.




```
String myString = "Hi There";
   String firstHalf =
      myString.substring(0, myString.length() / 2);
   String lastHalf =
      myString.substring(myString.length() / 2);
```

Q2

Java Class Library

- *class library*: collection of useful classes
- ***Java Class Library (JCL)*** is a set of classes that are part of the JDK and documented in the Java API (Application Programming Interface)
- These classes are not part of the Java language per se, but we rely on them heavily
- We've already used the `System`, `Scanner`, and `String` classes which are part of the JCL
- Other class libraries can be obtained through third party vendors, or you can create your own class library

JCL Packages

- Classes in the Java Class Library are organized into *packages*
- Example packages:

Package	Purpose
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities

- These packages are described in detail in Java API on Java's website (see jGRASP Help > Java API)

The import Declaration

- When you want to use a class from a package, you could use its fully qualified name (no import statement required)

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

- Or you can *import* the class and just use the class name

```
import java.util.Scanner; //at beginning of file
. . .
      Scanner scan = new Scanner(System.in);
```

- To import all classes in a package, you can use the * wildcard character

```
import java.util.*;
```

  - Not generally good practice; better to name each class used (as required by Checkstyle standard rules)

The import Declaration

- Why can I use the String class without importing its package (`java.lang`)?
  - The `java.lang` package is imported automatically!
  - It's as if the following line is always in a program:

```
import java.lang.*; // this would be redundant
```

- The `Scanner` class, on the other hand, is part of the `java.util` package, and therefore must be imported

```
import java.util.Scanner;
```

The Random Class

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudorandom numbers
- A `Random` object performs complicated calculations based on a *seed value* to produce a stream of pseudorandom values
- See RandomNumbersExample1.java

The Math Class

- The `Math` class is part of the `java.lang` package
- The `Math` class contains methods that perform various mathematical functions
- These include:
  - absolute value
  - square root
  - exponentiation
  - trigonometric functions
  - pseudorandom number generation

Math.random()

- The `random()` method in the Math class is a another way to generate pseudorandom numbers
- `Math.random()` returns a double value in the range from 0 to 1 which includes 0 but not 1; also written as the interval [0,1)
- Other ranges can be derived using multipliers and offsets
- See RandomNumbersExample2.java

Q3

The Math Class

- The methods of the `Math` class are *static methods* (also called *class methods*)
- Static methods should be invoked using the class name rather than an object reference (e.g., no object of the `Math` class is needed)

```
value = Math.cos(90) + Math.sqrt(delta);
```

- See Quadratic.java in the book

$$ax^2 + bx + c = 0 \qquad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- See Quadratic2.java which handles complex roots

Formatting Output

- You may want to format values in certain ways so that they can be presented properly

  8.2564634653 → 8.256

  1.08 → $1.08

- The `NumberFormat` class: formats values as currency or percentages
- The `DecimalFormat` class: formats values based on a pattern (including $ and %)
- Both are in the `java.text` package

Formatting Output

- The `NumberFormat` class has static methods that return a formatter object

```
getCurrencyInstance()

getPercentInstance()
```

- Each formatter object has a method called `format` that returns a string with the specified information in the appropriate format
- See PriceChange.java

We will use DecimalFormat (described next)

Formatting Output

- The `DecimalFormat` class can be used to format both floating point and integer values in various ways
- The constructor of the `DecimalFormat` class takes a String that represents the pattern for formatting of number
- The `format` method is called on the `DecimalFormat` object to return a `String` representing the formatted value

Formatting Output

- For example: We could round a double value to three significant decimal places by creating a `DecimalFormat` object with the pattern `"#.###"`

```
import java.text.DecimalFormat;
. . .
DecimalFormat df = new DecimalFormat("#.###");
double val = 4.123456789;
System.out.println(df.format(val));
```

Output:

```
4.123
```

Formatting Output

- Examples of some useful patterns

`"#,###.0#####"` – large amount with commas, up to six significant decimal places, and always at least one decimal place; 0 is returned as 0.0; 12.0 is returned as 12.0

`"#,###.######"` – large amount with commas and up to six significant decimal places; 12.0 is returned as 12; 123.456 is returned as 123.456

`"$#,##0.00"` – large amount of dollars with commas; 12345.6789 is returned as $12,345.68

`"$#,##0.00;($#,##0.00)"` – large amount of dollars with commas; negative values returned in parentheses

`"#.##%"` – a percentage with two significant decimal places; 0.123456 is returned as 12.35%

DecimalFormatExamples.java CylinderVolume.java $v = \pi r^2$

HalfEvenRoundingExamples.java

Default Rounding in Java

- Java uses ***half-even rounding*** for formatting - rounds toward the "nearest neighbor" unless both neighbors are equidistant, in which case, it rounds toward the even neighbor; also known as “bankers rounding”.

  If a number cannot be represented exactly in binary (e.g. 12.345), what appears to be equidistant is not (i.e., digit to be rounded is not exactly 5). So 12.345 rounded to two decimal places is 12.35 as it should be according to its exact decimal representation shown below from the jGRASP Numeric Viewer

DecimalFormat pattern "#.##"	
Value	Rounds to:
12.374	12.37
12.376	12.38
12.375	12.38
12.125	12.12
12.345	12.35

12.34500000000000639488462184090167284011840820312 5

Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type plus the reserved word `void`:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`void`	`Void`

Wrapper Classes

- The following declaration creates an `Integer` object which represents the integer 40 as an object

```
Integer age = new Integer(40);
```

- A primitive variable of type `int` type would not have methods; however, type `Integer`, the class, does have methods – for example:
  - byteValue(): returns the corresponding byte value
  - doubleValue(): returns the corresponding double value
  - toString(): returns the corresponding String value

Autoboxing and Unboxing

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;
int num = 42;
obj = num;
```

  - Creates the appropriate `Integer` object

- The reverse conversion (called *unboxing*) also occurs automatically as needed

```
num = obj;
```

Wrapper Classes

- Wrapper classes have useful static methods and constants
  - For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE`: the smallest and largest `int` values

    `Integer.MAX_VALUE is` 2147483647

    `Integer.MIN_VALUE is` -2147483648
  - The `Integer` and `Double` classes contain methods to convert a number stored in a `String` to a numeric value :

```
int i = Integer.parseInt("1234");
double x = Double.parseDouble("12.34");
```

(These two methods are used frequently!)

AutoBoxingAndMethodsExamples.java