

3. Using Classes and Objects

- Objectives - when we have completed this set of notes, you should be familiar with:
 - object creation and reference types
 - 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 *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("A book");
```

- 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

- Object variables are *reference variables*; they don't hold the object; they hold a 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...

Primitive Type: **num1** 52

Reference Type: **name1** [memory
address] → "Steve Jobs"

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("Intro to Computing");
```



Calls a *constructor* in the String class, which is a special method that sets up the String object

- 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 both:

```
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:

Before:

num1

38

num2

96

`num2 = num1;`

After:

num1

38

num2

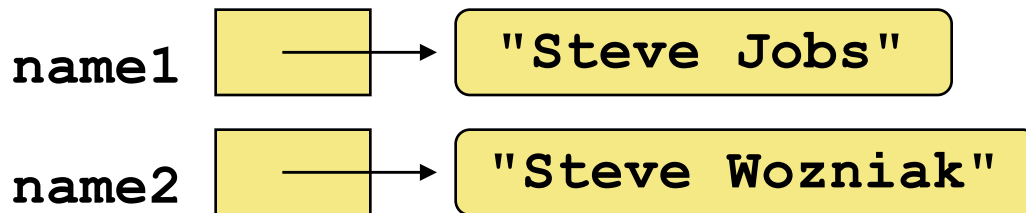
38

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

Reference Assignment

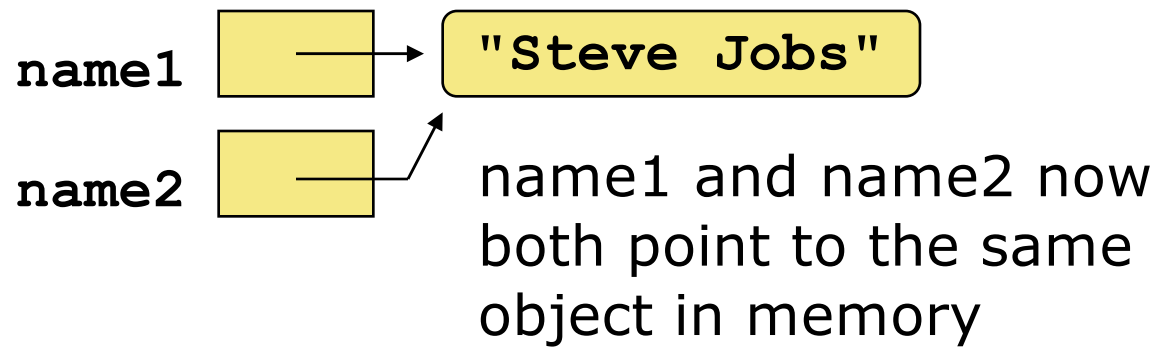
- For object references, assignment copies the address:

Before:



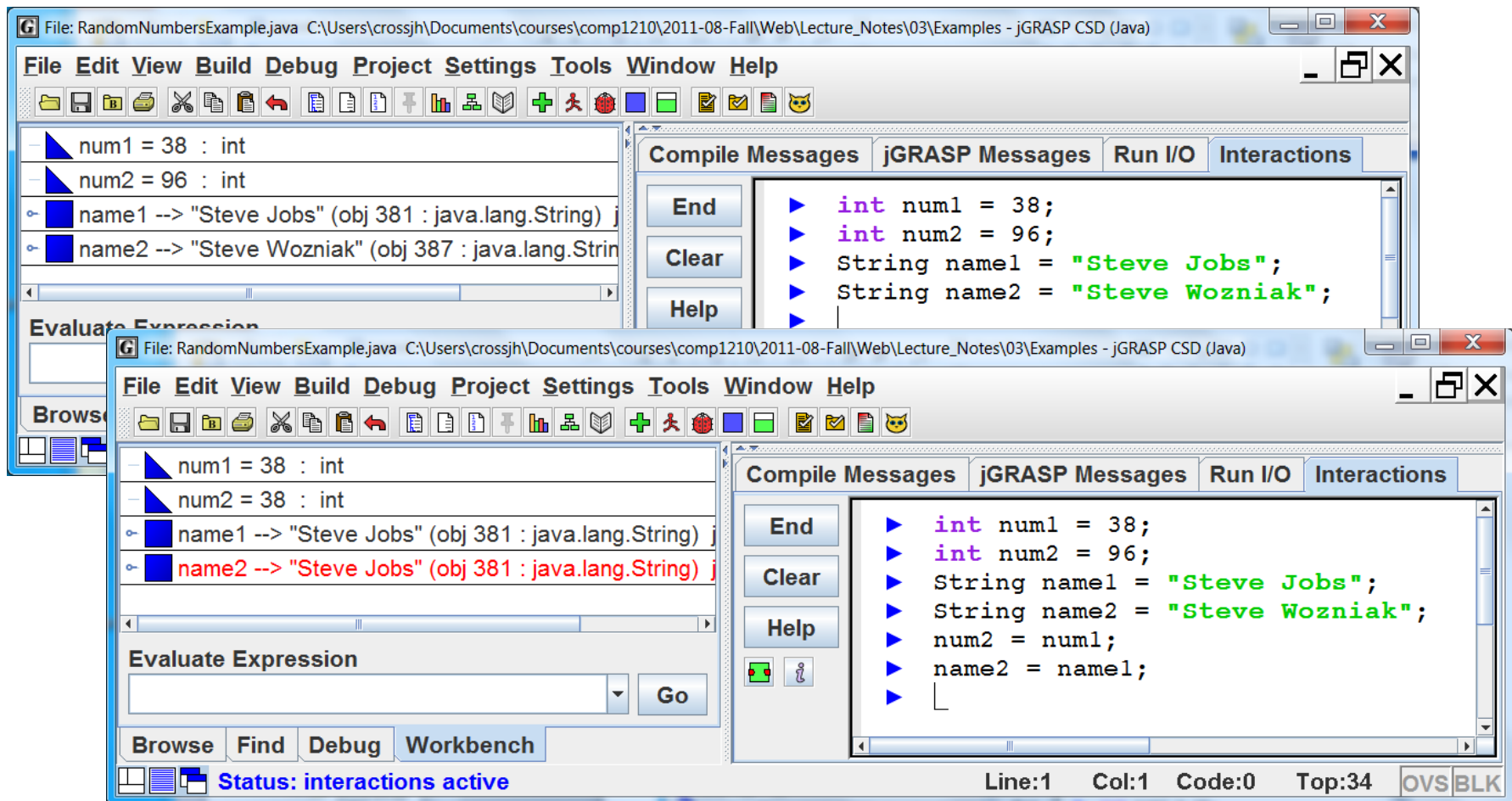
`name2 = name1;`

After:



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
- * other subtleties will be discussed in Ch 4

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

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("Intro to Computing I");
```

```
title = "Intro to Computing I";
```

- Each string literal (enclosed in double quotes) represents a `String` object
- All other reference types use of the `new` operator for object creation.

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("I", "1");
```

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

```
title.replace("e", "t");
```

- You probably meant to do this:

```
title = title.replace("e", "t");
```

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

H	e	l	l	o
0	1	2	3	4

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

H	i		T	h	e	r	e
0	1	2	3	4	5	6	7

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

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);
```

H	e	l	l	o
0	1	2	3	4

- 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);
```

H	i		T	h	e	r	e
0	1	2	3	4	5	6	7

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 them yourself (Chapter 4)

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 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](#)

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

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- See [Quadratic2.java](#) which handles complex roots
- `Math.random()` returns a double in range $[0,1)$

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
- Both are part of 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](#)

Formatting Output

- The `DecimalFormat` class can be used to format a floating point value in various ways
- For example, you can specify that the number should be “rounded” to three decimal places
 - Java uses ***half-even rounding*** for formatting
(Rounds toward the “nearest neighbor” unless both neighbors are equidistant, in which case, round toward the even neighbor; also known as “bankers rounding”. **Java uses this rounding mode for all floating point arithmetic.**)
- The constructor of the `DecimalFormat` class takes a string that represents a pattern for the formatted number
- See [CylinderVolume.java](#)

$$v = \pi r^2 h$$

Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

<u>Primitive Type</u>	<u>Wrapper Class</u>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>
<code>void</code>	<code>Void</code>

Wrapper Classes

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

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

- If age was an `int` type, it would not have methods
 - `byteValue()`: returns the corresponding byte value
 - `doubleValue()`: returns the corresponding double value

Wrapper Classes

- Wrapper classes also have useful static methods
- For example, the `Integer` class contains a method to convert an integer stored in a `String` to an `int` value:

```
num = Integer.parseInt(str);
```

- The wrapper classes often contain useful constants as well
 - For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest `int` values

```
Integer.MAX_VALUE
```

Autoboxing

- *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;
```