3. Using Classes and Objects

- Objectives - when we have completed this chapter, 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
  - GUI components and containers
  - GUI images

Review: Primitive Types

- Recall that a variable can be used to **store** a primitive type:
  - `int number;`
    - sets aside 32 bits of storage for an integer called number
  - `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: 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;
```

  - And *initialized (or assigned) with the **new** operator*:

```
title = new String("A book");
```

  - Or both *declared* and *initialized*:

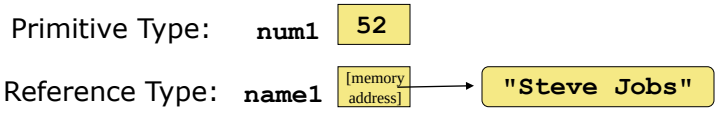
```
String team = new String("Red Sox");
```

  - 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' that store contents then reference variables are suitcases that contain an address that 'points' to the location of the contents.
- Represented graphically…



Creating Objects

- Declaration does not create an object.
  - Sets aside space for the **memory address** that title will hold

```
String title;
```

  - The placeholder memory address can be set to `null` to indicate that no String object has been created, which allows the program to check for the existence of the object.

```
title = null;
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 - - nextInt() in Scanner reads user input
  - invoked using the *dot operator* (.)
  - A method may *return* a value:

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

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

  - Method may accept *parameters* (input):

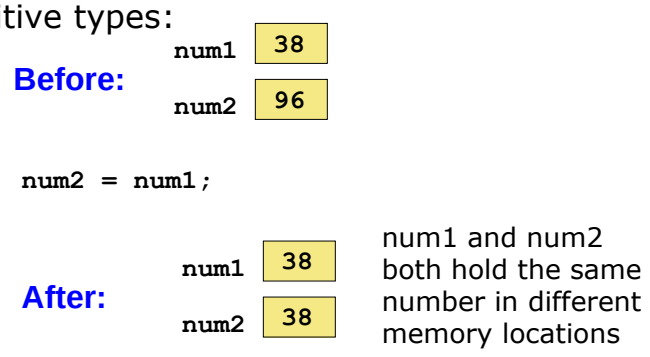
```
myScan.useDelimiter(",");
```

  - Or both:

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

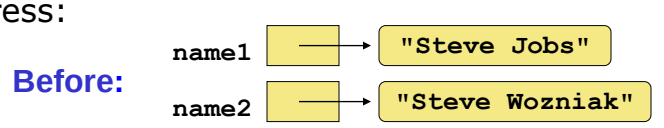

Assignment Revisited

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

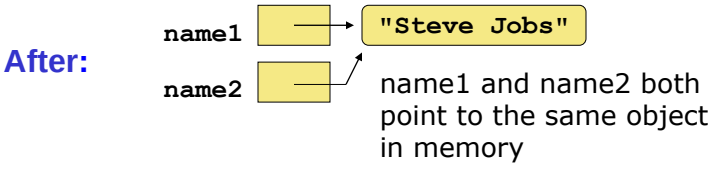


Reference Assignment

- For object references, assignment copies the address:



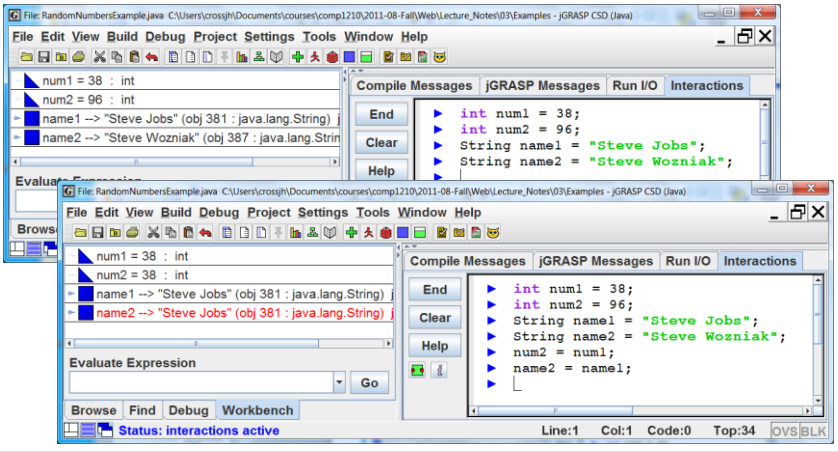

```
name2 = name1;
```



name1 and name2 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
  * 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.

```
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 require the use of the `new` operator for object creation.

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

- The following appears 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

- You can get a particular character from a String using the charAt method (given the index of the character)
- 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!)

    `01234567`
- See StringExample2.java

Q

Class Libraries

- *class library*: collection of useful classes
- ***Java standard class library*** is part of any Java development environment (documented in the Java API – see jGRASP Help > Java API)
- These classes are not part of the Java language per se, but we rely on them heavily
- Various classes we've already used (`System`, `Scanner`, `String`) are part of the Java standard class library
- Other class libraries can be obtained through third party vendors, or you can create them yourself (Chapter 4)

Packages

- Classes in the Java standard 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 (also 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 statememt 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; // top of source code
. . .
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; classes in different packages can have the same name and the compiler may select the wrong one

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

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 seemingly random values
- See RandomNumbersExample.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

The Math Class

- The methods of the `Math` class are *static methods* (also called *class methods*)
- Static methods can be invoked through the class name – no object of the `Math` class is needed

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

- See Quadratic.java in the book

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

- We discuss static methods further in Chapter 6

Q

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 know 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

Wrapper Classes

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

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);
```

- 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;
```


Graphical Applications

- Except for the applets seen in Chapter 2, the example programs we've explored thus far have been text-based
- They are called *command-line applications*, which interact with the user using simple text prompts
- Let's examine some Java applications that have graphical components
- These components will serve as a foundation to programs that have true graphical user interfaces (GUIs)

GUI Components

- A *GUI component* is an object that represents a screen element such as a button or a text field
- GUI-related classes are defined primarily in the `java.awt` and the `javax.swing` packages
- The *Abstract Windowing Toolkit* (AWT) was the original Java GUI package
- The *Swing* package provides additional and more versatile components
- Both packages are needed to create a Java GUI-based program

GUI Containers

- A *GUI* ***container*** is a component that is used to hold and organize other components
- A ***frame*** is a container that is used to display the GUI components of a Java application
- A frame is displayed as a separate window with a title bar – it can be repositioned and resized on the screen as needed
- A ***panel*** is a container that cannot be displayed on its own but is used to organize other components
- A panel may be added to another container (e.g., panel or frame) - - but remember that only a frame can be displayed

GUI Containers

- A GUI container can be classified as either heavyweight or lightweight
- A *heavyweight container* is one that is managed by the underlying operating system
- A *lightweight container* is managed by the Java program itself
- Occasionally this distinction is important
- A frame is a heavyweight container and a panel is a lightweight container

Labels

- A *label* is a GUI component that displays a line of text
- Labels are usually used to display information or identify other components in the interface
- Let's look at a program that organizes two labels in a panel and displays that panel in a frame
- See WarEagleFrame.java
- This program is not interactive, but the frame can be repositioned and resized

Nested Panels

- Containers that contain other components make up the *containment hierarchy* of an interface
- This hierarchy can be as intricate as needed to create the visual effect desired
- The following example nests two panels inside a third panel – note the effect this has as the frame is resized
- See SidePanels.java

Images

- Images are often used in a programs with a graphical interface
- Java can manage images in both JPEG and GIF formats
- As we've seen, a `JLabel` object can be used to display a line of text
- It can also be used to display an image
- That is, a label can be composed of text, and image, or both at the same time

Images

- The `ImageIcon` class is used to represent an image that is stored in a label
- The position of the text relative to the image can be set explicitly
- The alignment of the text and image within the label can be set as well
- See Pictures.java

GUI Wrap Up

- The code for graphical objects gets complicated
- Optimally, you don't just want it all in the main method; you want to separate the GUI code in a separate file
  - You'll learn how to do this next week
- Consider the examples in Chapter 3 "practice" for making more "useful" GUIs described in Chapter 4