More About Classes: Instance Methods

Chapter 6

Chapter Contents

Chapter Objectives

6.1 Introductory Example: Modeling Temperatures

6.2 Designing a Class

6.3 Implementing Class Attributes

6.4 Implementing Class Operations

6.5 Graphical/Internet Java: Raise the Flag

Chapter Objectives

- Look at how to build classes as types
- Study instance methods
  - contrast with static (class) methods
- Place in context of real world object (temperature)
- Implement attribute (instance) variables
- Explain importance of encapsulation and information hiding

Chapter Objectives

- Build a complete class to model temperatures
- Describe, give examples for
  - constructors, accessor methods, mutator methods, converter methods, utility methods
- Investigate graphics programming
- Look at artificial intelligence topics

Classes

- Generally used to describe a group or category of objects
  - attributes in common
- Java <u>class</u> used as a repository for static methods used by other classes
- Now we will create a class that serves as a <u>type</u>
  - from which objects are created
  - contains <u>instance methods</u>

6.1 Introductory Example: Modeling Temperatures

- Problem
  - Temperature Conversion
  - Fahrenheit, Celsius, Kelvin
- Preliminary Analysis
  - Attributes of temperature
    - number of degrees
    - scale
  - We seek a <u>type</u> which will …
    - hold all attributes and …
    - provide methods for manipulating those attributes

Object-Centered Design

Objects	Type	Kind	Name
program			
screen	`Screen`	varying	`theScreen`
prompt	`String`	constant	
temperature	`Temperature`	varying	`temp`
keyboard	`Keyboard`	varying	`theKeyboard`
Fahrenheit equivalent	`Temperature`	varying	
Celsius equivalent	`Temperature`	varying	
Kelvin equivalent	`Temperature`	varying	

Operations

- Display a string on **`theScreen`**
- Read a **`Temperature`** from **`theKeybard`**
- Determine Fahrenheit equivalent of **`Temperature`**
- Determine Celsius equivalent
- Determine Kelvin equivalent
- Display a **`Temperature`** on **`theScreen`**

Algorithm

1. Declare **`theScreen`**, **`theKeyboard`**, **`temp`**
2. Send **`theScreen`** message to display prompt
3. Send **`temp`** a message, ask it to read value from **`theKeyboard`**
4. Send **`theScreen`** a message to display
   - Fahrenheit, Celsius, Kelvin equivalents

Coding

- Note source code, Figure 6.2 in text
- Assumes existence of the class **`Temperature`**
- Note calls to **`Temperature`** methods
  `.read(theKeyboard)`
  `.inFahrenheit()`
  `.inCelsius()`
  `.inKelvin`

6.2 Designing a Class

- For a class, we must identify
  - Behavior, operations applied to class objects
  - Attributes, data stored to characterize a class object
- These are “wrapped together” in a class declaration

Class Declaration

- Syntax:

```
class className
{
  Method definitions
  Field Declarations
}
```

- Method definitions are as described in earlier chapters
- Field declarations are of variables and constants

External and Internal Perspectives

- External Perspective
  - observer from outside the program
  - views internal details
- Internal Perspective
  - object carries within itself ability to perform its operations
  - object autonomy

`Temperature` Behavior

- Define myself <u>implicitly</u>
  - initialize degrees, scale with default values
- Read value from a **`Keyboard`** object and store it within me
- Compute Fahrenheit, Celsius, Kelvin temperature equivalent to me
- Display my degrees and scale using a **`Screen`** object

Additional Behaviors Desired

- Define myself <u>explicitly</u> with degrees, scale
- Identify my number of degrees
- Identify my scale
- Increase, decrease my degrees by a specified number
- Compare myself to another `Temperature` object
- Assign another `Temperature` value to me

Temperature Attributes

- Review the operations
- Note information each requires
- Temperature has two attributes

1. my degrees
2. my scale

Implementing Class Attributes

- Stand alone class declared in a separate file: **`Temperature.java`**
- Specify variables to hold the attributes

```
double myDegrees;
char   myScale;
```

  - called the instance variables, data members, or fields

Encapsulation

- Wrap the attribute objects in a **`class declaration`**

```
class Temperature
{
  double myDegrees;
  char   myScale;
}
```

The class **`Temperature`** <u>encapsulates</u> **`myDegrees`** and **`myScale`**

- Use the class declaration as a type for declare actual objects

```
Temperature todaysTemp = new Temperature();
```

Information Hiding

- Attribute variables can be accessed directly
  `todaysTemp.myScale = 'Q'; // ???`
- We wish to ensure <u>valid</u> values only
- Solution is to “hide” the information to direct outside access

```
class Temperature
{
  private double myDegrees;
  private char   myScale;
}
```

It is good programming practice to hide all attribute variables of a class by specifying them as private

Class Invariants

- Important to identify restrictions on values of attributes
  - minimum, maximum temp
  - **`myScale`** limited to F, C, or K
- Specify with boolean statements in comments

```
. . .
private char myScale; // 'F', 'C', or 'K'
```

Class Invariants

- Helpful to specify static (class) constants

```
class Temperature
{


public final static double
  ABS_ZERO_F = -459.67;
  ABS_ZERO_C = -273.15;
  ABS_ZERO_K = 0.0;
...
```

All objects of type **`Temperature`** share a single instance of these values

6.4 Implementing Static Operations

- Use instance methods
- Contrast:

Static (Class) Methods	Instance (Object) Methods
Declared with keyword **`static`**	No **`static`** modifier used
Shared by all objects of class	Each class has its own copy
Invoke by sending message to the <u>class</u>	Invoked by sending message to class <u>object</u>

Instance Methods Categories

- Constructors
  - initialize attribute variables
- Accessors
  - retrieve (but not change) attribute variables
- Mutators
  - change attribute variable values
- Converters
  - provide representation of an object in a different type
- Utilities
  - used by other methods to simplify coding

`Temperature` Output: a Convert-to-`String` Method

- `print()` and `println()` methods used
  - display a value whose type is `Object` or any class that extends `Object`
  - send object message to convert itself to `String` using `toString()` (a converter)
- Thus `theScreen.print(todaysTemp)` "asks" `todaysTemp` to return a `String` representation of itself
- Note source code Figure 6.3 in text

Constructor Methods

`Temperature temp1 = new Temperature`

- Initial values for **`myDegree`** and **`myScale`** are `0` and **`NULL`**, respectively
  - Better to give them valid values
- Constructor method used to give these values
  - default-value constructor
  - explicit value constructor

Default Value Constructor

- Whenever a **`Temperature`** object is declared, this specifies initial values

```
public   Temperature()
{ myDegrees = 0.0;
  myScale = 'C'; }
```

- Note:
  - no return type (not even void)
  - name of constructor method must be same as name of class

Explicit-Value Constructors

- Useful to initialize values at declaration time
- Explicit value constructor
  - uses parameters to initialize **`myDegrees`** and **`myScale`**
  - Note source code Figure 6.5 in text
- Constructor invoked by …

```
Temperature todaysTemp =
        new Temperature(75,'F');
```

Method Names and Overloading

- Now we have two methods with the same name
  - but different numbers of parameters
- Two or more methods with same name called “overloading”
  - compiler determines which method to use
  - based on number and/or types of arguments in call

Utility Methods

- Class **`Temperature`** methods need check for validity of incoming or changing values of variables
  - **`myDegrees`** must be greater than absolute zero
  - **`myScale`** must be one of **`'C'`**, **`'K'`**, or **`'F'`**
- Utility method provided **`isValidTemperature()`**

A Utility Method: `fatal()`

- `isValidTemperature()` handles only the incoming parameters
  - not the attribute variables
- If they are wrong the `fatal()` method is called (note source code Figure 6.7)
  - displays diagnostic message
    - method where problem detected
    - description of problem
  - terminates execution

Static vs. Instance Methods

- Note that `toString()` and the constructor methods are instance methods
  - `isValidTemperature()` and `fatal()` are static methods
- Instance method:
  - invoked by message sent to instance of a class
- Static method:
  - invoked by message sent to the class itself

Static vs. Instance Methods

- Static methods
  - may <u>only</u> access static variables, constants, and static methods
  - access only <u>static</u> declared items
- Instance methods
  - may access <u>both</u> instance <u>and</u> static variables, constants, methods
- Objects have their own distinct copies of
  - <u>instance</u> variables, constants, methods
- Objects share the same copy of
  - <u>static</u> variables, constants, methods

Class Design Pointers

- Most variables should be declared as attribute variables
- If a method needs to access attribute variables
  - then define it as a <u>instance</u> method
- If a method does <u>not</u> need to access static variables
  - make it a <u>static</u> (class) method
  - pass information to it via parameters or declared class attributes

Accessor Methods

- Methods that allow program to <u>retrieve</u> but not modify class attributes
- Example:

```
public double getDegrees()
  {    return myDegrees;    }
```

Mutator Methods

- Input into a **`Temperature`** object
- Desired command:
  `todaysTemp.read(theKeyboard);`
  - reads a number, a character from keyboard
  - stores them in proper variables
- This is a method that changes values of attribute variables
  - thus called a "mutator"

Managing the Input

- Need for strategy to handle invalid inputs from user
  - will return boolean value to indicate validity of inputs
- Note the source code, Figure 6.9 of text – observe differences from constructor
  - values come from theKeyboard instead of parameters
  - returns boolean value instead of generating fatal error

Conversion Methods

- A temperature object should be able to compute any scale equivalent of itself
  - method returns appropriate value
  - based on current value of **`myScale`**
- Note source code, Figure 6.10
  - **`result`** initialized to null
  - method constructs a **`Temperature`** value for **`result`**
  - return statement makes result value returned

Raising/Lowering a `Temperature`

- We need a method which enables the following command
  ```
  tuesTemp = monTemp.raise(4.5);
                   // or .lower()
  ```
- The return value would be an object of the same scale, different **`myDegrees`**
- Method should use **`isValidtemperature()`** to verify results
  - if invalid results, uses the **`fatal()`** utility
- Note source code Figure 6.11

Comparing `Temperature` Values

- We cannot use
  `if (monTemp < tuesTemp) ...`
- We must use something like
  `if monTemp.lessThan(tuesTemp) ...`
- View source code Figure 6.12, note:
  - must convert to proper scale for comparison
  - then simply return results of comparison of **`myDegrees`** with the results of parameter's **`getDegrees()`** method

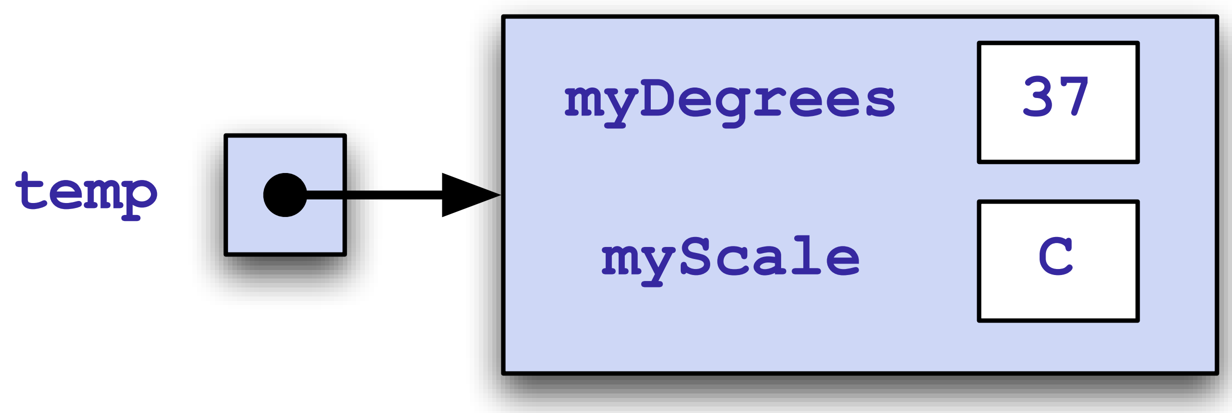
Similar strategy for the **`.equals()`** method

Alternate Comparison Strategy

- Note the duplicate code in the **`.lessThan()`** and **`.equals()`** methods
- Write a single method **`.compareTo()`** which returns **-1**, **0**, or **+1** signifying **<**, **==**, or **>**
- Rewrite **`.lessThan()`** and **`.equals()`** to call the **`.compareTo()`** and decide the equality/inequality based on **-1**, **0**, or **+1**

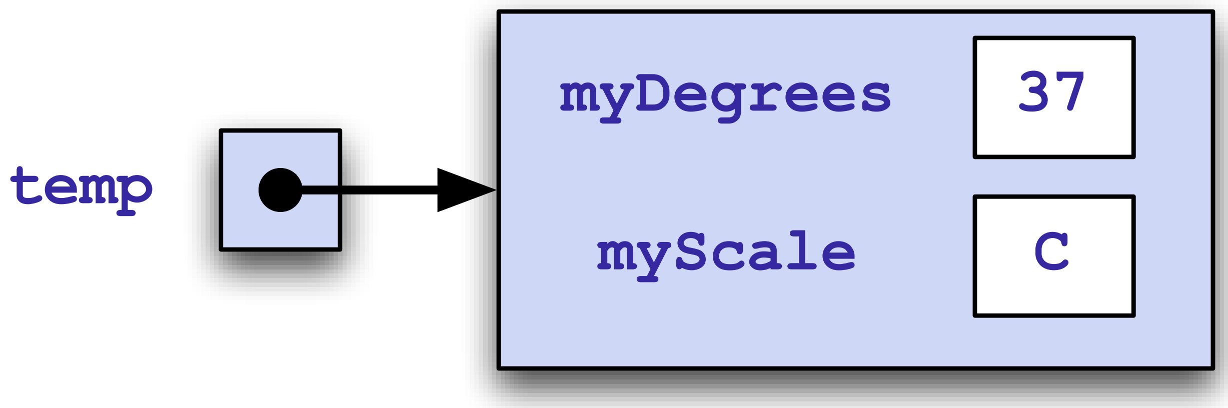
Reference-type Declaration

- "Reference" is another word for "address"
  `Temperature temp = new Temperature(37,'C');`
- The variable temp really holds the <u>address</u> for the memory location allocated by the **`new`** command



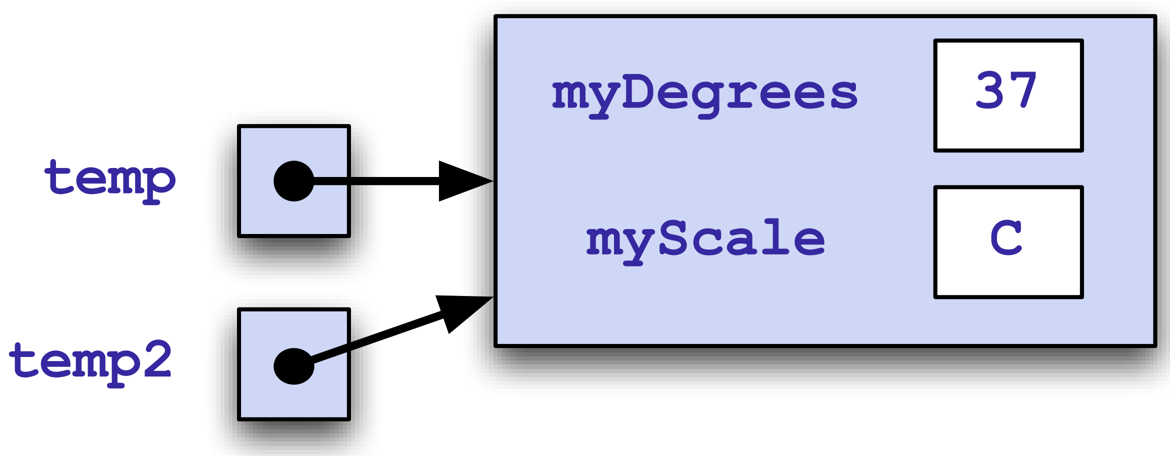
Handles

- **temp** is the only way to access the **Temperature** object
  - it has no name of its own
  - **temp** is the handle for the object it references



Reference Type Copying

- Consider the following two statements:
```
Temperature temp = new Temperature(37,'C');
Temperature temp2 = temp;
```
- Note: declaration of **`temp2`** did not use the **`new`** command
  - a new object did not get created
  - we merely have two handles for one object



Reference Type Copying

- At times we need to create another object, not just another pointer
  - create a **copy** method
  - returns a distinct Temperature object, equal to (a clone of) itself

```
public Temperature copy()
{
return new Temperature(myDegrees,myScale);
}
```

- Invoked as shown:
  Temperature newTemp = oldTemp.copy();

Reference Type Copying

- Note simplicity of this `copy` method
  - all attribute variables happen to be primitive types
- If attribute variables were, themselves, reference types
  - our version would make only handles to the actual attribute variable objects
  - this called “shallow” copy
- For truly distinct, “deep” copy
  - each reference type attribute variable must be copied separately

Class Organization

- Note source code of entire class, Figure 6.17
- Standard practice
  - begin class with constants class provides
  - follow with constructors, accessors, mutators, converters, utilities
  - place attribute variable declarations last

Class Interface

- Benefits of private attribute variables
  - forces programs to interact with class object through its public methods
  - public operations thought of as the "interface"
- Design the interface carefully
  - gives stability to the class
  - even though implementation of methods changes, use of the class remains unchanged

6.5 Graphical/Internet Java: Raise the Flag

- A **ClosableFrame** Class
  - provided in **ann.gui** package
  - we will build classes that extend this class

```
class DrawingDemo extends ClosableFrame
{ public static void main(String [] args)
  { DrawingDemo myGUI = new DrawingDemo();
    myGUI.setVisible(true);
  }
}
```

Sends the new object a message to make itself visible

Creates a new instance

Inheritance

- **`DrawingDemo`** class inherits all attributes of **`CloseableFrame`**
  - variables and constants
  - behaviors (methods)
- Sample methods of CloseableFrame
  - set the frame title
  - set colors
  - set size
  - access width, height
  - set visibility

Painting

- Top-level containers contain intermediate containers
  - called panes or panels
- <u>Content pane</u> is most important
  - used to group, position components
- Note source code, Figure 6.18 in text
  - `main` method now also creates a `DrawingPane` and specifies the size

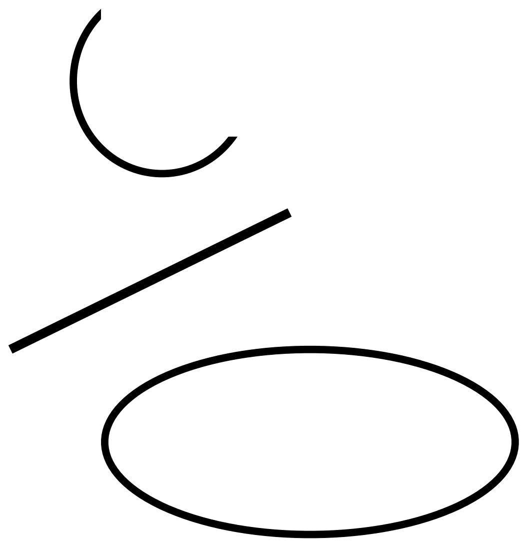
Methods in `DrawingPain()`

- Use subclasses of **`JPanel`**
  - constructor sets background to white
- **`paintComponent`**()
  - painting of **`Swing`** components <u>must</u> be performed by a method with this name
- This is where statements that do the actual painting reside

```
public void paintComponent(Graphics pen)
{  /* statements to do painting */  }
```

`Graphics` Class Methods

- Sample graphics methods …
- drawArc
- drawLine
- drawOval



- drawString   This is a string

Dutch Flag GUI Application

- Note source code Figure 6.19
- Uses the **`paintComponent()`** method
  - draws two rectangles
  - red filled on top
  - blue filled on bottom
  - middle stripe is original white background

Dutch Flag Applet

- Source code Figure 6.20
- Many Swing components used in both applets and applications
- Modifications:
  - This class extends **`JApplet`** instead of **`CloseableFrame`**
  - change **`main`** method to **`init()`**

Part of the Picture: Artificial Intelligence

- Recently (5/97) a computer program defeated a world chess champion
- Construction of game playing programs is known as “artificial intelligence” or AI for short
- Definition of AI is difficult
  - intelligent behavior is complex
  - styles of programming AI are diverse

Intelligence

- A chess playing program is "intelligent" in a very <u>narrow</u> domain
- General human intelligence is demonstrated in a <u>wide range</u> of behaviors

AI Topics

- Reasoning and problem solving
- Memory of things in our world
- Motion and manipulation of objects (robotics)
- Perception
  - computer vision, speech recognition
- Language processing
  - understanding and generation
  - translation
- Learning from past experiences

Which of these can the chess playing computer do?

AI Programming Techniques

- Heuristic search
  - search through choices and consequences
- Logic programming
  - represent knowledge in well defined format
  - perform logic inferences on it

AI Programming Techniques

- Expert systems
  - encode knowledge from an expert in some domain
- Neural networks
  - model the way the brain works
  - use highly interconnected simple processes

Example: JackDice Game

- Similar to blackjack
- Roll two dice, sum the values
- Continue rolling as desired
- Come as close to 21 without going over
- Simulate this “intelligent” activity with a Java program – see driver program, Figure 6.21

Strategies for JackDice

- Scaled-down expert system
  - encode “knowledge” from expert players
- Examples of expert knowledge
  - always accept the first roll (never risks passing 21)
  - randomly decide whether to go on or not (???)
  - take more risks if you are behind in a game
  - play conservative if you are ahead