More About Classes – Instance Methods

Chap.6
Study Sections 6.1 – 6.4

Representing More Complex Objects

1

Starting the Move from OCD to OOP

Characteristics of OOP:

Encapsulation: classes combine data and operations into self-contained units

Inheritance: classes can be defined as extensions of other classes

Later

Polymorphism: Poly–"many"; morph–"forms" different classes may implement the same operations in different ways

2

Extending OCD

1. **Formulate behavior needed to solve problem**
2. **Identify objects in problem**
   For each object for which there is no type in the language:
   - ***Design and build a class to represent it.***
3. **Identify operations needed in problem**
   For each operation not provided in the language:
   - ***Design and build a method to perform it.***
   - ***Store the method in a class responsible for providing the operation.***

   Labs 4 & 5
4. **Organize objects and operations in an algorithm**

3

Classes vs. Objects

Primary purpose of a class isn't to encapsulate methods.

It is to describe objects.

Classes are used to create objects.
We call objects *instances* of a class.

If a class is a:	An object is a:
factory	product
blueprint	building

4

- A class is not an object; it *describes* an object.
- The *type* of an object is a class.
- *One class* can be used to create *many objects*. Each is an *instance* of the class.
- Each instance gets its *own* (instance) methods and (instance) variables that are described in the class.

5

Example

What goes into a class usually arises out of a problem; so we begin with one.

Dealing with different systems of measurement is a nuisance and can be a very serious problem (e.g., Mars Climate Orbiter — 1999).

Here (and in the text) we consider the problem of processing temperatures in various scales (Fahrenheit, Celsius, Kelvin).

6

Preliminary Analysis

We should be able to represent common objects with a *single type*.

But temperature objects have two attributes:

- *magnitude* (a double), and
- *scale* (a character)

When an object cannot be directly represented by any of the available types, *build a new type* (thus *extending the language*)!

7

Behavior

Our program should display a prompt for a temperature (in either Fahrenheit, Celsius, or Kelvin) on the screen, read that temperature from the keyboard, compute and display corresponding temperatures in all three systems along with a descriptive label.

8

Objects

Our program should display a prompt for a temperature (in either Fahrenheit, Celsius, or Kelvin) on the screen, read that temperature from the keyboard, compute and display corresponding temperatures in all three systems along with a descriptive label.

9

Operations

Our program should display a prompt for a temperature (in either Fahrenheit, Celsius, or Kelvin) on the screen, read that temperature from the keyboard, compute and display corresponding temperatures in all three systems along with a descriptive label.

10

Algorithm

1. Ask *the screen* to display *a prompt* for *the temperature*.

2. Ask the keyboard to read *the temperature*.

3. Compute *Fahrenheit, Celsius and Kelvin versions* of that temperature.

4. Ask *the screen* to display *the three versions*, plus an *informative label* on *the screen*.

11

Representing Objects

Object	**Java Type**	**Name**
the program	--	--
a prompt	`String`	--
the input temperature	???	*temp*
the screen	`Screen`	*theScreen*
the keyboard	`Keyboard`	*theKeyboard*
a label	`String`	--

We need to create a `Temperature` type.

12

Performing Operations

Operation	Library?	Name
Display a string	`ann.easyio`	`print()`
Read a temperature	???	*read*()
Compute fahrenheit	???	*inFahrenheit*()
Compute celsius	???	*inCelsius*()
Compute kelvin	???	*inKelvin*()
Display a temperature	???	*println*()???

We need to create a **`Temperature`** type that has input and output operations and converters from one scale to another (and perhaps other operations).


```
/** TemperatureConverter.java
 *  Input:  a temperature value (e.g., 0 C).
 *  Output: that temperature in Celsius, Fahrenheit and Kelvin.
 */

import ann.easyio.*;

class TemperatureConverter
{
 public static void main(String [] args)
 {
   Screen theScreen = new Screen();
   Keyboard theKeyboard = new Keyboard();
   Temperature temp = new Temperature();
   boolean inputOK;
   for (;;)
   {
     theScreen.print("\nTo perform a temperature conversion, enter\n"
                    + " a temperature (e.g., 0 C). Enter 0 A to stop: ");
     inputOK = temp.read(theKeyboard);

     if (!inputOK) break;

     theScreen.println( "--> " + temp.inFahrenheit()
                      + " = "  + temp.inCelsius()
                      + " = "  + temp.inKelvin() );
   }
 }
}
```

The Driver Code
(that we'd like to be able to write)

Using Classes
(External Perspective)

The program doesn't know how to:

- initialize the data stored for each temperature object
- read and write a temperature object
- convert a temperature object into various forms

The <u>temperature object itself is responsible</u> for these things.

The <u>program sends messages</u> to temperature objects to execute methods for these operations.

Building Classes

Two phases:
1. Design
2. Implementation

"I can do it myself" principle

Best done from an internal perspective.
It puts you in the role of the object as you build its class:

```
"I am a Screen."
"I am a String."
"I am a Temperature object."
```

Design:

- Describe class' *behavior* — *operations* applied to class objects
- Identify its *attributes* — *data* stored to characterize a class object

Implementation:

- *Encapsulate* operations and attributes in a class declaration:

```
class ClassName
{
  // Method definitions for built-in operations
  // Data member declarations for data
}
```

17

Design of Temperature Class

Behavior

Operations needed to solve problem:

- Construct myself with initial default values
- Read a temperature value from the keyboard and store it in myself
- Display myself on the screen
- Convert myself to Fahrenheit
- Convert myself to Celsius
- Convert myself to Kelvin

Other operations to increase reusability

- Construct myself with initial specified values
- Tell my degrees
- Tell my scale
- Increase myself by a given number of degrees
- Decrease myself by a given number of degrees
- Compare myself to another Temperature object
- Assign another Temperature value to me

Note the internal (*"I can do it myself"*) perspective

18

Attributes

To find them, go through the operations and find information each of them requires.
If needed by several, make it an attribute.

- **my degrees**
- **my scale**

Note: Initial list may be incomplete.
But we can always add more later as we build (and maintain) the class

19

Implementation of Temperature Class

Implement the attributes first.

Declare variables to hold the attributes. They determine the *state* of the object over time. They are called *attribute variables* or *instance variables*, or *fields*, or *data members*.

```
// Temperature.java

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

Not finished here yet!

Pretending that *we are* the Temperature object, we begin the name of each attribute with the prefix **my** to reinforce the *internal perspective*.

20

Information Hiding

To prevent direct access to these attribute variables from outside the class, we declare them to be private:

```
// Temperature.java
class Temperature
{
  private double myDegrees;
  private char myScale;
}
```

This is known as *information hiding*.

21

Why?

- They could be assigned invalid values (e.g., an 'X' scale).
- Software must often be updated. Updating a class may require that its data members be modified (e.g., make **myScale** a string), renamed, replaced, removed, added.
- If a program accesses class data members directly, then that program "breaks" if they are changed. This increases software maintenance costs! Programs that use a class should still work after the class has been updated.

22

The principle of *information hiding* dictates that a class designer always distinguish between the:

- *External interface* to a class: Make *public* only those things that a class user really needs to know.
- *Internal implementation* of the class: Hide all other details by making them *private.*

23

This is encapsulation!

```
// Temperature.java
class Temperature
{
  // Definitions of methods for reading,
     writing, & converting temperatures...
     (Most are public.)

  // Private declarations of attribute
     variables (also called instance
     variables, fields, or data members)
  private double myDegrees;
  private char myScale;
}
```

24

Kinds of Methods

Static methods

- One definition is *shared* by all the objects in the class
- Declared with the keyword **`static`**
- *Invoked with a message to the class*

Instance methods

- Each object has its *own copy* of the method
- Declared by default (i.e., not static)
- *Invoked with a message to an instance of the class; i.e. to an object*

Static methods

- *Can access only static items in a class.*
- *Use if it doesn't need to access the instance variables of the class.*

Instance methods

- *Can access both instance and static items in a class.*
- *Most of the methods we write for a class are instance methods.*

Kinds of Instance Methods

Constructors	Initialize instance variables
Accessors	Retrieve values of instance variables
Mutators	Change values of instance variables
Convertors	Produces a different representation of an object
Utilities	Used to simplify other methods or to avoid redundant code

Output

- Add an output operation early so we can check other operations.
- When **`print()`** and **`println()`** are asked to display an object, they send it a message to *convert* itself to a **`String`** using its **`toString()`** method. So we need only add this method to our class:

```
/** toString converter (used by print(), println(), ...
 *  Return a string representation of myself.
 */
public String toString()
{
  return myDegrees + " " + myScale;
}
```

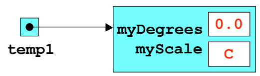
Constructor Methods

Users should be able to construct new Temperature objects in two ways:

- default initial values

*Calling the **default** constructor*

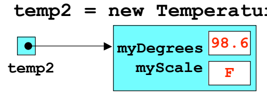
```
Temperature temp1 = new Temperature();
```



- explicitly specified values

*Calling the **explicit-value** constructor*

```
Temperature temp2 = new Temperature(98.6, 'F');
```



This is the role of the class constructors.

29

Default-Value Constructor

```
/** Default-value constructor.
 *  Postcondition: myDegrees == 0.0 && myScale == 'C'.
 */
public Temperature()
{
  myDegrees = 0.0;
  myScale = 'C';
}
```

- The name of a constructor is always the *name of the class* (in this case **`Temperature()`**).
- Since it returns nothing, a constructor has *no return type* (not even **`void`**).
- Specification is usually a postcondition describing the state of the constructed object.

30

Explicit-Value Constructor

```
/** Explicit-value constructor.
 *  Receive:       double degrees, char scale
 *  Precondition:  scale is one of 'f', 'F', 'c', 'C', 'k', or 'K'
 *                 degrees is a valid number of degrees for scale.
 *  Postcondition: myDegrees == degrees &&
 *                 myScale == (uppercase) scale.
 */

public Temperature(double degrees, char scale)
{
  // we'll add more later

    myDegrees = degrees;
    myScale = Character.toUpperCase(scale);

  // more later
}
```

31

pp. 284-5, 222-3

Overloading Methods

Our constructors are two different methods with the same name **`Temperature`**. Such a name is said to be overloaded.

When is this permitted? First, a definition: A method's name and the list of its parameter types is known as the method's signature.

> *The name of a method can be overloaded provided no two definitions of the method have the same signature.*

32

Incremental (Unit) Testing

We can now begin a driver program that will be used to test our class (and add to it as we develop more methods):

Output:

```
temp1 = 0.0 C
temp2 = 98.6 F
```

```
import ann.easyio.*;

class TempDriver
{
  public static void main(String [] args)
  {
    Screen theScreen = new Screen();
    Keyboard theKeyboard = new Keyboard();
    Temperature temp1 = new Temperature(),
                temp2 = new Temperature(98.6, 'F');
    theScreen.println(  "temp1 = " + temp1 +
                      "\ntemp2 = " + temp2 );
  }
}
```

33

Class Invariants

PROBLEM: *Objects must maintain the integrity of their internal data*. However, the explicit-value constructor can assign invalid values to the class' instance variables. (e.g., 0 X, -10 K)

A *class invariant* is a condition that must be true throughout the class. The most important one is that the *instance variables always contain valid values.*

Example: For the Temperature class:

- the scale can only be 'C', 'F', or 'K'
- the degrees should be greater than absolute zero

34

Any method that modifies instance variables – called a *mutator* method – must ensure that the class invariant will be true.

We can express our class invariant as a boolean expression:

```
       myScale == 'C'
         && myDegrees >= ABSOLUTE_ZERO_CELSIUS
    || myScale == 'F'
         && myDegrees >= ABSOLUTE_ZERO_FAHRENHEIT
    || myScale == 'K'
         && myDegrees >= ABSOLUTE_ZERO_KELVIN
```

It's good practice to include the class invariant in the class' opening documentation.

35

We begin by adding definitions of the absolute-zero constants at the beginning of our class:

```
class Temperature
{
  public final static double
          ABSOLUTE_ZERO_FAHRENHEIT = -459.67,
          ABSOLUTE_ZERO_CELSIUS    = -273.15,
          ABSOLUTE_ZERO_KELVIN     =    0.0;
  . . .
}
```

Why public?

They're probably useful in other temperature programs, so we make them accessible outside the class (e.g., **`Temperature.ABSOLUTE_ZERO_KELVIN`**)

Why static??????

36

Static (Class) vs. Instance Data

Static data

- One definition is *shared* by all the objects in the class
- Declared with the keyword `static`
- *Constants* are usually static; e.g., absolute zero constants

Instance data

- Each object has its *own copy*
- Declared by default (i.e., not static)
- *Variables* are usually instance ; e.g., `myDegrees, myScale`

37

Utility Methods

To help the explicit-value constructor and other mutators with checking a complex class invariant, we can add a *utility method* like the following to our class to save having to write an if statement each time:

```
public static boolean isValidTemperature(double degrees,
                                         char scale)
{
  if (scale == 'C' || scale == 'c')
     return degrees >= ABSOLUTE_ZERO_CELSIUS;
  else if (scale == 'F' || scale == 'f')
     return degrees >= ABSOLUTE_ZERO_FAHRENHEIT;
  else if (scale == 'K' || scale == 'k')
     return degrees >= ABSOLUTE_ZERO_KELVIN;
  else
     return false;
}
```

Why static?

38

If the class invariant fails, we might call another utility method like the following to display an error message and halt execution:

User need not know about this method

Doesn't need to access instance variables

```
private static void fatal(String methodName,
                          String diagnostic)
{
  System.err.println("\n*** " + methodName + ": "
                     + diagnostic);
  System.err.flush();
  System.exit(1);
}
```

Ensure that output gets to the screen.

Halt execution

Note: `fatal()` is useful in other classes, so it was added to `ann.util.Controller`.

39

Safe Explicit-Value Constructor

```
/** Explicit-value constructor.
 *  Receive:       double degrees, char scale
 *  Precondition:  scale is one of 'f', 'F', 'c', 'C', 'k', or 'K'
 *                 degrees is a valid number of degrees for scale.
 *  Postcondition: myDegrees == degrees &&
 *                 myScale == (uppercase) scale.
 */
public Temperature(double degrees, char scale)
{
  if (isValidTemperature(degrees, scale))
  {
    myDegrees = degrees;
    myScale = Character.toUpperCase(scale);
  }
  else
    fatal("Temperature(degrees, scale)",
          "invalid args: " + degrees + scale);
}
```

40

Accessor Methods

```
/** degrees accessor
 *  Return:  myDegrees
 */
public double getDegrees()
{
  return myDegrees;
}

/** scale accessor
 *  Return:  myDegrees
 */
public char getScale()
{
  return myScale;
}
```


Mutator Methods

```
/** raise a temperature
 *  Receive: double amount
 *  Precondition: myDegrees + amount is valid for myScale
 *  Return:  the raised Temperature object
 */
public Temperature raise(double amount)
{
  double newDegrees = myDegrees + amount;

  if (!isValidTemperature(newDegrees, myScale))
    fatal("raise(double)", newDegrees + " "
             + myScale + " is not a valid temperature");
  return new Temperature(newDegrees, myScale);
}
```

Temperature-lowering method is similar.


```
/** read a temperature
 * Receive: Keyboard object in
 * Input:   inDegrees (double), inScale (char)
 * Return:  true if valid temperature read, else false
 * Postcondition: myDegrees == inDegrees &&
 *                myScale == inScale (if valid)
 */
public boolean read(Keyboard in)
{
  double inDegrees = in.readDouble();
  char inScale     = in.readChar();

  if (isValidTemperature(inDegrees, inScale))
  {
    myDegrees = inDegrees;
    myScale   = Character.toUpperCase(inScale);
    return true;
  }
  else
    return false;
}
```

Usage:

```
boolean ok = temp.read(theKeyboard);
if (ok)
{  . . .
```


An alternative (ala Lab 6):

```
import java.util.StringTokenizer;
  . . .
public void read(Keyboard in)
{
  Keyboard.EatWhiteSpace();
  String tempStr = in.readLine();
  StringTokenizer parser = new StringTokenizer(tempStr);
  if (parser.countTokens() != 2)
    fatal("read(Keyboard in)",
          " -- Bad format for temperature");
  //else
  double inDegrees = Double.parseDouble(parser.nextToken());
  char inScale = parser.nextToken().charAt(0);
  if (!isValidTemperature(inDegrees, inScale))
    fatal("read(Keyboard in)",
          " -- Bad format for temperature");
  //else
  myDegrees = inDegrees;
  myScale   = Character.toUpperCase(inScale);
}
```

Conversion Methods

We've written one converter -- `toString()`. Now we look at one of the converters from one scale to another; the others are similar.

```
/** Fahrenheit converter
 *  Return: the Fahrenheit equivalent to myself
 */
public Temperature inFahrenheit()
{
  Temperature result = null;

  if (myScale == 'F')
    result = new Temperature(myDegrees, 'F');
  else if (myScale == 'C')
    result = new Temperature(myDegrees*1.8 + 32.0, 'F');
  else if (myScale == 'K')
    result = new Temperature((myDegrees-273.15)*1.8 + 32.0,'F');

  return result;
}
```

45

Comparison Methods

We can't compare new types with:

```
if (temp1 < temp2)
```

Unlike C++, Java doesn't allow operator overloading.

Thus, we must write methods for comparison.

46

```
/** less-than
 * Receive: another Temperature object otherTemp
 * Return:  true if-f I am less than otherTemp
 */
public boolean lessThan(Temperature otherTemp)
{
  Temperature localTemp = null;

  if (myScale == 'C')
    localTemp = otherTemp.inCelsius();
  else if (myScale == 'F')
    localTemp = otherTemp.inFahrenheit();
  else if (myScale == 'K')
    localTemp = otherTemp.inKelvin();

  return myDegrees < localTemp.getDegrees();
}
```

Other comparison methods are basically the same.

47

```
public int compareTo(Temperature otherTemp)
{
  Temperature localTemp = null;
  if (myScale == 'C')
    localTemp = otherTemp.inCelsius();
  else if (myScale == 'F')
    localTemp = otherTemp.inFahrenheit();
  else if (myScale == 'K')
    localTemp = otherTemp.inKelvin();

  if (myDegrees < localTemp.getDegrees())
    return -1;
  else if (myDegrees > localTemp.getDegrees())
    return 1;
  else
    return 0;
}
```

Another way to compare

```
public boolean lessThan(Temperature otherTemp)
{
  return compareTo(otherTemp) < 0;
}
```

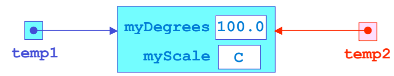
Other comparison methods are basically the same.

48

Copy (or Clone) Operation

Picture what the following code will do:

```
Temperature temp1 = new Temperature(100, 'C');
Temperature temp2 = temp1;
```



`Temperature` is a reference type; "reference" = "address";

Value of **`temp1`** is an *address* that refers ("points") to the actual temperature object; we call it a *handle*.

Second declaration gives **`temp2`** the same value (i.e., the *same address*, so it refers to the same temperature object; changing one changes both!

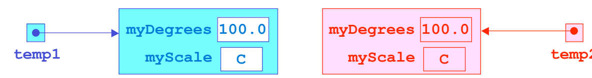
This is called the *aliasing problem*.

To avoid it, add a copy operation to the class:

```
/** copy operation
 *  Return:  A distinct copy of myself
 */
public Temperature copy()
{
  return Temperature(myDegrees, myScale);
}
```

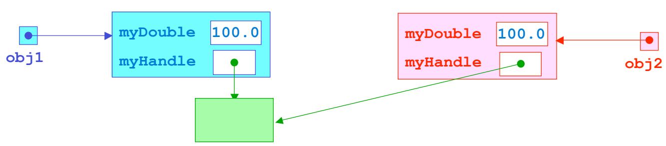
Now picture:

```
Temperature temp1 = new Temperature(100, 'C');
Temperature temp2 = temp1.copy();
```

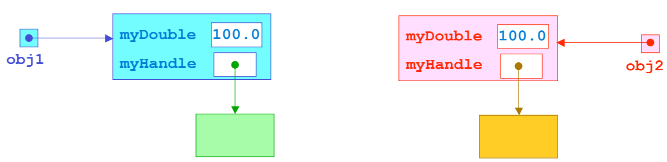


`temp2` refers to a *distinct copy* of the object referred to by **`temp1.`**

But . . .
we still have an aliasing problem when some data members are of a reference type with no copy operation:



This is a *shallow copy*; we need a *deep copy*.



Have every class provide a copy operation.