Chapter 5

More About Objects and Methods

- Programming with Methods
- Static Methods and Static Variables
- Designing Methods
- Polymorphism
- Constructors
- Information Hiding Revisited
- Packages

The "`this.`" Operator

- *`this.`* refers to the object that contains the reference
- Methods called in an object definition file do not need to reference itself
- You may either use "`this.`", or omit it, since it is presumed
- For example, if `answerOne()` is a method defined in the class `Oracle`:

```
public class Oracle
{
   ...
   //One way to invoke the answerOne method defined
   //in this file: this.answerOne();

   //Another way is to omit "this."
   answerOne();  //"this." is presumed
   ...
}
```

When an Object Is Required

- Methods called *outside* the object definition require an object to precede the method name
- For example:

```
Oracle myOracle = new Oracle();
//myOracle is not part of the definition code
//for Oracle
...
//dialog is a method defined in Oracle class
myOracle.dialog();
...
```


Static Methods

- Some methods do work but do not need an object
  - » For example, methods to calculate area: just pass the required parameters and return the area
- Use the class name instead of an object name to invoke them
- For example
  - » `CircleFirstTry` is a class with methods to perform calculations on circles: `CircleFirstTry.area(myRadius);`
  - » Notice that the the method invocation uses "*`className.`*" instead of "*`circleObject.`*"
- Also called *class methods*

Static Methods

- Declare static methods with the *`static`* modifier, for example:

```
public static double area(double radius)
  ...
```

- Since a static method doesn't need a calling object, it cannot refer to a (nonstatic) instance variable of the class.
- Likewise, a static method cannot call a nonstatic method of the class (unless it creates an object of the class to use as a calling object).

Uses for Static Methods

- Static methods are commonly used to provide libraries of useful and related functions
- Examples:
  - » the `Math` class
    - – automatically provided with Java
    - – functions include `pow`, `sqrt`, `max`, `min`, etc.
    - – see the next slide for more details
  - » `SavitchIn` defines methods for keyboard input
    - – not automatically provided with Java
    - – functions include `readLineInt`, `readLineDouble`, etc.
    - – see the appendix

The **`Math`** Class

- Includes constants `Math.PI` (approximately 3.14159) and `Math.E` (base of natural logarithms which is approximately 2.72)
- Includes three similar static methods: `round`, `floor`, and `ceil`
  - » All three return whole numbers (although they are type `double`)
  - » **`Math.round`** returns the whole number nearest its argument

`Math.round(3.3)` returns `3.0` and `Math.round(3.7)` returns `4.0`

  - » **`Math.floor`** returns the nearest whole number that is equal to or less than its argument

`Math.floor(3.3)` returns `3.0` and `Math.floor(3.7)` returns `3.0`

  - » **`Math.ceil`** (short for ceiling) returns the nearest whole number that is equal to or greater than its argument

`Math.ceil(3.3)` returns `4.0` and `Math.ceil(3.7)` returns `4.0`

Static Variables

- The `StaticDemo` program in the text uses a static variable:
  `private static int numberOfInvocations = 0;`
- Similar to definition of a named constant, which is a special case of static variables.
- May be public or private but are usually private for the same reasons instance variables are.
- Only one copy of a static variable and it can be accessed by any object of the class.
- May be initialized (as in example above) or not.
- Can be used to let objects of the same class coordinate.
- Not used in the rest of the text.

Wrapper Classes

- Used to wrap primitive types in a class structure
- All primitive types have an equivalent class
- The class includes useful constants and static methods, including one to convert back to the primitive type

Primitive type	Class type	Method to convert back
`int`	`Integer`	`intValue()`
`long`	`Long`	`longValue()`
`float`	`Float`	`floatValue()`
`double`	`Double`	`doubleValue()`
`char`	`Character`	`charValue()`

Wrapper class example: `Integer`

- Declare an `Integer` class variable:

```
Integer n = new Integer();
```

- Convert the value of an `Integer` variable to its primitive type, `int`:

```
int i = n.intValue();//intValue returns
  an int
```

- Some useful `Integer` constants:
  - » `Integer.MAX_VALUE` - the maximum integer value the computer can represent
  - » `Integer.MIN_VALUE` - the smallest integer value the computer can represent

Wrapper class example: `Integer`

- Some useful `Integer` methods:
  - » `Integer.valueOf("123")` to convert a string of numerals to an integer
  - » `Integer.toString(123)` to convert an `Integer` to a `String`
- The other wrapper classes have similar constants and functions
- See the text for useful methods for the class `Character`

Usage of wrapper classes

There are some important differences in the code to use wrapper classes and that for the primitive types

Wrapper Class

- variables contain the *address* of the value
- variable declaration example:
  ```
  Integer n = new Integer();
  ```
- variable declaration & init:
  ```
  Integer n = new Integer(0);
  ```
- assignment:
  ```
  n = new Integer(5);
  ```

Primitive Type

- variables contain the value
- variable declaration example:
  ```
  int n;
  ```
- variable declaration & init.:
  ```
  int n = 0;
  ```
- assignment:
  ```
  n = 99;
  ```

Designing Methods: Top-Down Design

- In pseudocode, write a list of subtasks that the method must do.
- If you can easily write Java statements for a subtask, you are finished with that subtask.
- If you cannot easily write Java statements for a subtask, treat it as a new problem and break it up into a list of subtasks.
- Eventually, all of the subtasks will be small enough to easily design and code.
- Solutions to subtasks might be implemented as private helper methods.
- Top-down design is also known as *divide-and-conquer* or *stepwise refinement*.

Programming Tips for Writing Methods

- Apply the principle of encapsulation and detail hiding by using the `public` and `private` modifiers judiciously
  - » If the user will need the method, make it part of the interface by declaring it `public`
  - » If the method is used only within the class definition (a *helper* method, then declare it `private`
- Create a `main` method with diagnostic (test) code within a class's definition
  - » run just the class to execute the diagnostic program
  - » when the class is used by another program the class's `main` is ignored

Testing a Method

- Test programs are sometimes called *driver* programs
- Keep it simple: test only one new method at a time
  - » driver program should have only one untested method
- If method A uses method B, there are two approaches:
- *Bottom up*
  - » test method B fully before testing A
- *Top down*
  - » test method A and use a *stub* for method B
  - » A *stub* is a method that stands in for the final version and does little actual work. It usually does something as trivial as printing a message or returning a fixed value. The idea is to have it so simple you are nearly certain it will work.

Overloading

- The same method name has more than one definition *within the same class*
- Each definition must have a different signature
  - » different argument types, a different number of arguments, or a different ordering of argument types
  - » The return type is **not** part of the signature and **cannot** be used to distinguish between two methods with the same name and parameter types

Signature

- the combination of method name and number and types of arguments, in order
- `equals(Species)` has a different signature than `equals(String)`
  - » same method name, different argument types
- *`myMethod(1)`* has a different signature than *`myMethod(1, 2)`*
  - » same method name, different number of arguments
- *`myMethod(10, 1.2)`* has a different signature than *`myMethod(1.2, 10)`*
  - » same method name and number of arguments, but different order of argument types

Gotcha: Overloading and Argument Type

- Accidentally using the wrong datatype as an argument can invoke a different method
- For example, see the `Pet` class in the text
  - » `set(int)` sets the pet's age
  - » `set(double)` sets the pet's weight
  - » You want to set the pet's weight to 6 pounds:
    - `set(6.0)` works as you want because the argument is type `double`
    - `set(6)` will set the *age* to 6, not the weight, since the argument is type `int`

Gotcha: Automatic Type Conversion and Overloading

- If Java does not find a signature match, it attempts some automatic type conversions, e.g. `int` to `double`
- An unwanted version of the method may execute
- In the text Pet example of overloading:
  What you want: name "Cha Cha", weight 2, and age 3
  But you make two mistakes:
  1. you reverse the age and weight numbers, and
  2. you fail to make the weight a type double.
  - » `set("Cha Cha", 2, 3)` does not do what you want
    - – it sets the pet's age = 2 and the weight = 3.0
  - » Why?
    - – `set` has no definition with the argument types `String`, `int`, `int`
    - – However, it does have a definition with `String`, `int`, `double`, so it promotes the last number, 3, to 3.0 and executes the method with that signature

Constructors

- A *constructor* is a special method designed to initialize instance variables
- Automatically called when an object is created using ***new***
- Has the same name as the class
- Often overloaded (more than one constructor for the same class definition)
  - » different versions to initialize all, some, or none of the instance variables
  - » each constructor has a different signature (a different number or sequence of argument types)

Defining Constructors

- Constructor headings do not include the word `void`
- In fact, constructor headings do not include a return type
- A constructor with no parameters is called a *default constructor*
- If no constructor is provided Java automatically creates a default constructor
  - » If *any* constructor is provided, then *no* constructors are created automatically

Programming Tip

- Include a constructor that initializes *all* instance variables
- Include a constructor that has no parameters
  - » include your own *default constructor*

Constructor Example from **`PetRecord`**

```
public class PetRecord
{
   private String name;
   private int age; //in years
   private double weight; //in pounds
. . .
   public PetRecord(String initialName)
   {
      name = initialName;
      age = 0;
      weight = 0;
   }
```

Initializes three instance variables: `name` from the parameter and `age` and `weight` with default initial values.

Sample use:
`PetRecord pet1 = new Pet("Eric");`

Using Constructors

- Always use a constructor after `new`
- For example, using the `Pet` class in text:
  `Pet myCat = new Pet("Calvin", 5, 10.5);`
  - » this calls the `Pet` constructor with `String`, `int`, `double` parameters
- If you want to change values of instance variables after you have created an object, you must use other methods for the object
  - » you cannot call a constructor for an object after it is created
  - » `set` methods should be provided for this purpose

Information Hiding Revisited

- Using instance variables of a class type takes special care
- The details are beyond the scope of this text, but
  - » the problem in described in section 5.4/page275
  - » Appendix 6 covers some of the details
- The problem stems from the fact that, unlike primitive types, object identifiers contain the object's address, not its value
  - » returning an object gives back the address, so the called method has direct access to the calling object
  - » the calling object is "unprotected" (usually undesirable)
  - » best solution: c*loning*, but that is beyond the scope of this book
- One solution: stick to returning primitive types (`int`, `char`, `double`, `boolean`, `etc.`) or `String`

Packages

- A way of grouping and naming a collection of related classes
  - » they serve as a *library* of classes
  - » they do not have to be in the same directory as your program
- The first line of each class in the package must be the keyword `package` followed by the name of the package:
  ```
  package mystuff.utilities;
  ```
- To use classes from a package in a program put an `import` statement at the start of the file:
  ```
  import mystuff.utilities.*;
  ```
  - » note the `".*"` notation

Package Naming Conventions

- Use lowercase
- The name is the file pathname with subdirectory separators ("\" or "/", depending on your system) replaced by dots
- For example, if the package is in a file named "utilities" in directory "mystuff", the package name is:
  ```
  mystuff.utilities
  ```

Package Naming Conventions

- Pathnames are usually relative and use the `CLASSPATH` environment variable
- For example, if: `CLASSPATH=c:jdk\lib\mystuff`, and your file `utilities` is in `c:jdk\lib\mystuff`, then you would use the name: `utilities`
  - » the system would look in directory `c:jdk\lib\mystuff` and find the `utilities` package

Summary
Part 1

- A method definition can use a call to another method of the same class
- *static* methods can be invoked using the class name or an object name
- Top-down design method simplifies program development by breaking a task into smaller pieces
- Test every method in a program in which it is the only untested method

Summary
Part 2

- Each primitive type has a corresponding wrapper class
- *Overloading:* a method has more than one definition in the same class (but the number of arguments or the sequence of their data types is different)
  - » one form of polymorphism
- *Constructor:* a method called when an object is created (using *new*)
  - » *default constructor:* a constructor with no parameters