Chapter 4

Classes, Objects, and Methods

- Class and Method Definitions
- Information Hiding and Encapsulation
- Objects and Reference
- Parameter passing
Learn by Doing

- All programs in the book are available on the CD that comes with the book
- It is a good idea to run the programs as you read about them
- Do not forget that you will need the \texttt{SavitchIn.java} file for keyboard input
- Classes are used to define objects and provide methods to act on the objects
- Classes are also programs that declare these objects and process them to solve the problem
Classes

- *Class*—definition of a kind of object
- Like an outline or plan for constructing specific objects
  - see next slide or diagram in text
- Example: an Automobile class
  - Object that satisfies the *Automobile* definition *instantiates* the *Automobile* class
- Class specifies what kind of data objects of that class have
  - Each object has the same data items but can have different values
- Class specifies what methods each object will have
  - All objects of the same class have the exact same methods
Class Name: Automobile

Data:
- amount of fuel
- speed
- license plate

Methods (actions):
- increaseSpeed:
  How: Press on gas pedal.
- stop:
  How: Press on brake pedal.

First Instantiation:
- Object name: patsCar
- amount of fuel: 10 gallons
- speed: 55 miles per hour
- license plate: “135 XJK”

Second Instantiation:
- Object name: suesCar
- amount of fuel: 14 gallons
- speed: 0 miles per hour
- license plate: “SUES CAR”

Third Instantiation:
- Object name: ronsCar
- amount of fuel: 2 gallons
- speed: 75 miles per hour
- license plate: “351 WLF”
Objects

- Objects are variables that are named instances of a class
  - the class is their type
- Objects have both data and methods
- Both the data items and methods of a class are members of the object
- Data items are also called fields or instance variables
- Invoking a method means to call the method, i.e. execute the method
  - Syntax for invoking an object's method: the dot operator
    
    ```java
    object_Variable_Name.method()
    ```
  - `object_Variable_Name` is the calling object
Example: \textit{String Class}

- \textit{String} is a class
  - it stores a sequence of characters
  - its \texttt{length} method returns the number of characters

- Example: read characters typed in by user from the keyboard and output the number of characters entered
  
  \begin{verbatim}
  String userInput;
  userInput = SavitchIn.readLine();
  System.out.println(userInput.length());
  \end{verbatim}
Class Files

- Each Java class definition should be a separate file
- Use the same name for the class and the file, except add ".java" to the file name
- Good programming practice: Start the class (and file) name a capital letter and capitalize inner words upper case
  » e.g. MyClass.java for the class MyClass
- For now put all the classes you need to run a program in the same directory
Instance Variables

- The `SpeciesFirstTry` class has three instance variables: `name`, `population`, and `growthRate`:

  ```java
  public String name;
  public int population;
  public double growthRate;
  ```

- `public` means that there are no restrictions on how these instance variables are used.
- Later we’ll see that these should be declared `private` instead of `public`. 
Instantiating (Creating) Objects

- Syntax:
  
  ```java
  class_Name instance_Name = new class_Name();
  ```

- Note the keyword `new`

- For example, the text defines a class named `SpeciesFirstTry`

  ```java
  // instantiate an object of this class
  SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
  ```

- Public instance variables can be accessed using the dot operator:

  ```java
  SpeciesOfTheMonth.name = "Klingon ox";
  ```
Using Methods

- Methods are actions that an object can perform.
- To use a method you *invoke* or *call* it.

Example of a method call:

```
speciesOfTheMonth.writeOutput()
```

- **calling object**—tells which object will do the action
- **method name**—tells which action the object will perform
- **parameter list in parentheses**—parameters give info to the method, but in this example there are no parameters

- Two basic kinds of methods:
  - methods that return a single value
  - *void* methods that do some action other than returning a value
Return Type of Methods

- All methods require that the return type be specified.
- Return types may be:
  - A primitive data type, such as `char`, `int`, `double`, etc.
  - A class, such as `String`, `SpeciesFirstTry`, etc.
  - `void` if no value is returned.
- You can use a method anywhere where it is legal to use its return type, for example the `readLineInt()` method of `SavitchIn` returns an integer, so this is legal:
  
  ```java
  int next = SavitchIn.readLineInt();
  ```
**void** Method Example

- The definition of the `writeOutput` method of `SpeciesFirstTry`:

  ```java
  public void writeOutput() {
      System.out.println("Name = " + name);
      System.out.println("Population = " + population);
      System.out.println("Growth = " + growthRate + "\%"));
  }
  ```

- Assuming instance variables `name`, `population`, and `growthRate` have been defined and assigned values, this method performs an action (writes values to the screen) but does not return a value
Return Statement

- Methods that return a value must execute a `return` statement that includes the value to return

- For example:
  ```java
  public int getCount()
  {
    return count;
  }
  public int count = 0;
  ```
Method and Class Naming Conventions

Good Programming Practice

- Use verbs to name void methods
  - they perform an *action*
- Use nouns to name methods that return a value
  - they create (return) a piece of data, a *thing*
- Start class names with a capital letter
- Start method names with a lower case letter
The `main` Method

- A program written to solve a problem (rather than define an object) is written as a class with one method, `main`
- Invoking the class name invokes the `main` method
- See the text: `SpeciesFirstTryDemo`
- Note the basic structure:

```java
public class SpeciesFirstTryDemo {
    public static void main(String[] args) {
        <statements that define the main method>
    }
}
```
The Reserved Word **this**

- The word **this** has a special meaning for objects.
- It is a *reserved* word, which means you should not use it as an identifier for a variable, class or method.
  - Other examples of reserved words are `int`, `char`, `main`, etc.
- **this** stands for the name of the calling object.
- Java allows you to omit **this**.
  - It is automatically understood that an instance variable name without the keyword **this** refers to the calling object.
Example Using `this`

- Using the same example as for the `void` method, but including the keyword `this`:

```java
public void writeOutput()
{
  System.out.println("Name = " + this.name);
  System.out.println("Population = " + this.population);
  System.out.println("Growth rate = " + this.growthRate + "%");
}
```

- `this` refers to the name of the calling object that invoked the `writeOutput` method
Local Variables and Blocks

- A block (a compound statement) is the set of statements between a pair of matching braces (curly brackets).
- A variable declared inside a block is known only inside that block:
  - it is local to the block, therefore it is called a local variable
  - when the block finishes executing, local variables disappear
  - references to it outside the block cause a compile error
Local Variables and Blocks

- Some programming languages (e.g. C and C++) allow the variable name to be reused outside the local block
  - it is confusing and not recommended, nevertheless, it is allowed
- However, a variable name in Java can be declared only once for a method
  - although the variable does not exist outside the block, other blocks in the same method cannot reuse the variable's name
When and Where to Declare Variables

- Declaring variables outside all blocks but within the method definition makes them available within all the blocks.

**Good programming Practice:**
- Declare variables just before you use them.
- Initialize variables when you declare them.
- Do not declare variables inside loops.
  - It takes time during execution to create and destroy variables, so it is better to do it just once for loops.
- It is ok to declare loop counters in the *Initialization* field of *for* loops, e.g.,
  ```java
  for(int i=0; i < 10; i++)...
  ```
  - The *Initialization* field executes only once, when the *for* loop is first entered.
Passing Values to a Method: Parameters

- Some methods can be more flexible (therefor useful) if we pass them input values.
- Input values for methods are called *passed* values or *parameters*.
- Parameters and their data types must be specified inside the parentheses of the heading in the method definition.
  - these are called *formal* parameters.
- The calling object must put values of the same data type, in the same order, inside the parentheses of the method invocation.
  - these are called *arguments*, or *actual* parameters.
Parameter Passing Example

//Definition of method to double an integer
public int doubleValue(int numberIn)
{
    return 2 * numberIn;
}
//Invocation of the method... somewhere in main...
...
int next = SavitchIn.readLineInt();
System.out.println("Twice next = " +
doubleValue(next));

● What is the formal parameter in the method definition?
   » numberIn

● What is the argument in the method invocation?
   » next
Pass-By-Value: 
**Primitive Data Types as Parameters**

- When the method is called, the *value* of each argument is *copied* (assigned) to its corresponding formal parameter.
- The number of arguments must be the same as the number of formal parameters.
- The data types of the arguments must be the same as the formal parameters and in the same order.
- Formal parameters are initialized to the values passed.
- Formal parameters are local to their method.
- Variables used as arguments cannot be changed by the method.
  » the method only gets a copy of the variable's value.
Summary of Class Definition Syntax

/****************************
 * Class description
 * Preconditions (see the text)
 * Postconditions (see the text)
 ****************************/

public class Class_Name
{
    //Method definitions of the form
    /****************************
     * Method description
     ****************************/
    public returnType Method_Name(type1 parameter1, ...)
    {
        <statements defining the method>
    }

    <Instance variable definitions - accessible to all methods>
}

Information Hiding and Encapsulation

- Cornerstones of Object Oriented Programming (OOP)
- Both are forms of abstraction

Information hiding
- Protect data inside an object
- Do not allow direct access

Encapsulation
- Use classes and objects
- Objects include both data items and methods to act on the data
public and private

public
- any other class or program can directly access or change a public instance variable
- any other class or program can invoke a public method

private
- only a method in the same class can access a private instance variable
- only a method in the same class can invoke a public method

Instance variables should be private to prevent inappropriate changes.
Accessors and Mutators

- **accessor methods**—public methods that allow instance variables to be read

- **mutator methods**—public methods that allow instance variables to be modified
  - Mutator methods should always check to make sure that changes are appropriate.
  - Providing mutator methods is much better than making instance variables public because a method can check to make sure that changes are appropriate.
Precondition and Postcondition Comments

- efficient and standard way to tell what a method does
- **precondition**—states conditions that must be true before method is invoked
- **postcondition**—tells the effect of a method call
- Example:

```java
/**
 * Precondition: years is a nonnegative number
 * Postcondition: Returns the projected population after the specified number of years
 */
```

- Note that the terms preconditions and postconditions are not always used, particularly if the only postcondition describes the return value of the method.
Assertion Checks

- **assertion**—statement that should be true if there are no mistakes in the program
- Preconditions and postconditions are examples of assertions.
- Can use assert to see if assertion is true.
- Syntax:
  
  ```java
  assert Boolean_Expression;
  ```
- Example:
  
  ```java
  assert n >= limit;
  ```
- If assertion is false when checked, the program ends and an error message is printed.
- Assertion checking can be turned on and off.
  - The exact way to enable or disable assertions depends on your development environment.
A programmer who uses the class can only access the instance variables indirectly through public methods and constants.
Formalized Abstraction: ADTs

**ADT: Abstract data type**
- An Object-Oriented approach used by several languages
- A term for *class* implementation
  - a container for both data items and methods to act on the data
- Implements information hiding and encapsulation
- Provides a public *user interface* so the user knows how to use the class
  - descriptions, parameters, and names of its methods
- Implementation:
  - private instance variables
  - method definitions are usually public but always hidden from the user
  - the user cannot see or change the implementation
  - the user only sees the interface
Sound Complicated?

Not really! Just create classes as previously described, except:

- Use the `private` modifier when declaring instance variables
- Do *not* give the user the class definition file
- *Do* give the user the interface - a file with just the class and method descriptions and headings
  - the headings give the names and parameters of the methods
  - it tells the user how to use the class and its methods
  - it is all the user needs to know
UML Class Diagrams

Automobile

- fuel: double
- speed: double
- license: String

+ increaseSpeed(double howHardPress): void
+ stop(double howHardPress): void

Graphical notation to summarize some of the main properties of a class
Variables: Class Type vs. Primitive Type

What does a variable hold?
  » It depends on the type of type, primitive type or class type

- A primitive type variable holds the value of the variable

- Class types are more complicated
  » they have methods and instance variables

- A class type variable holds the memory address of the object
  » the variable does not actually hold the value of the object
  » in fact, as stated above, objects generally do not have a single value and they also have methods, so it does not make sense to talk about its "value"
Assignment with Variables of a Class Type

```java
klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);
earth = klingon;
earth.set("Elephant", 100, 12);
System.out.println("earth:");
earth.writeObject();
System.out.println("klingon:");
klingon.writeObject();
```

**What will the output be?**

(see the next slide)
Assignment with Variables of a Class Type

```java
klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);
earth = klingon;
earth.set("Elephant", 100, 12);
System.out.println("earth: ");
earth.writeObject();
System.out.println("klingon: ");
klingon.writeObject();
```

What will the output be?

**klingon and earth both print Elephant.**

**Why do they print the same thing?**

(see the next slide)
Assignment with Variables of a Class Type

klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);

earth = klingon;
earth.set("Elephant", 100, 12);

System.out.println("earth:");
earth.writeOutput();
System.out.println("klingon:");
klingon.writeOutput();

Why do they print the same thing?
The assignment statement makes earth and klingon refer to the same object.

When earth is changed to "Elephant", klingon is changed also.
Gotcha: Comparing Class Variables

- A class variable returns a number, but it is not its value.
- It returns the *memory address* where the object with that variable name is stored.

If two class variables are compared using `==`, it is the addresses, not the values that are compared! This is rarely what you want to do!

- Use the class's `.equals()` method to compare the *values* of class variables.
Example: Comparing Class Variables

```java
//User enters first string
String firstLine = SavitchIn.readLine();

//User enters second string
String secondLine = SavitchIn.readLine();

if(firstLine == secondLine)
//this compares their *addresses*
{
    <body of if statement>
}

if(firstLine.equals(secondLine))
//this compares their *values*
{
    <body of if statement>
}
```

Use `.equals` method (not the double-equals sign) to compare values
Pass the Address: 
Class Types as Method Parameters

- In the same way, class variable names used as parameters in a method call copy the argument's \textit{address} (not the value) to the formal parameter.
- So the formal parameter name also contains the address of the argument.
- It is as if the formal parameter name is an alias for the argument name.

\begin{center}
Any action taken on the formal parameter is actually taken on the original argument!
\end{center}

- Unlike the situation with primitive types, the original argument is not protected for class types!
Example: Class Type as a Method Parameter

```java
//Method definition with a DemoSpecies class parameter
public void makeEqual(DemoSpecies otherObject)
{
    otherObject.name = this.name;
    otherObject.population = this.population;
    otherObject.growthRate = this.growthRate;
}

//Method invocation
DemoSpecies s1 = new DemoSpecies("Crepek", 10, 20);
DemoSpecies s2 = new DemoSpecies();
s1.makeEqual(s2);
```

- The method call makes `otherObject` an alias for `s2`, therefore *the method acts on s2, the DemoSpecies object passed to the method!*

- This is *unlike* primitive types, where the passed variable cannot be changed.
Summary
Part 1

- Classes have instance variables to store data and methods to perform actions
- Declare instance variables to be private so they can be accessed only within the same class
- There are two kinds of methods: those that return a value and \textit{void}-methods
- Methods can have parameters of both primitive type and class type
Summary
Part 2

- Parameters of a primitive type work differently than those of a class type
  - primitive type parameters are call-by-value, so the calling object's variable is protected within the called method (the called method cannot change it)
  - class type parameters pass the address of the calling object so it is unprotected (the called method can change it)

- For similar reasons, the operators = and == do not behave the same for class types as they do for primitive types (they operate on the address of object and not its values)

- Therefore you should usually define an equals method for classes you define (to allow the values of objects to be compared)