

Chapter 4: Writing Classes

Presentation slides for

Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley

Presentation slides are copyright 2000 by John Lewis and William Loftus. All rights reserved.

Instructors using the textbook may use and modify these slides for pedagogical purposes.

Writing Classes



- **We've been using predefined classes. Now we will learn to write our own classes to define new objects**
- **Chapter 4 focuses on:**
 - **class declarations**
 - **method declarations**
 - **instance variables**
 - **encapsulation**
 - **method overloading**
 - **graphics-based objects**

Objects

- ⦿ **An object has:**

- *state* - descriptive characteristics
- *behaviors* - what it can do (or be done to it)

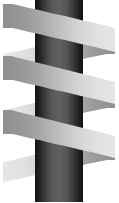
- ⦿ **For example, consider a coin that can be flipped so that its face shows either "heads" or "tails"**
- ⦿ **The state of the coin is its current face (heads or tails)**
- ⦿ **The behavior of the coin is that it can be flipped**
- ⦿ **Note that the behavior of the coin might change its state**

Classes

- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the `String` class is used to define `String` objects
- Each `String` object contains specific characters (its state)
- Each `String` object can perform services (behaviors) such as `toUpperCase`

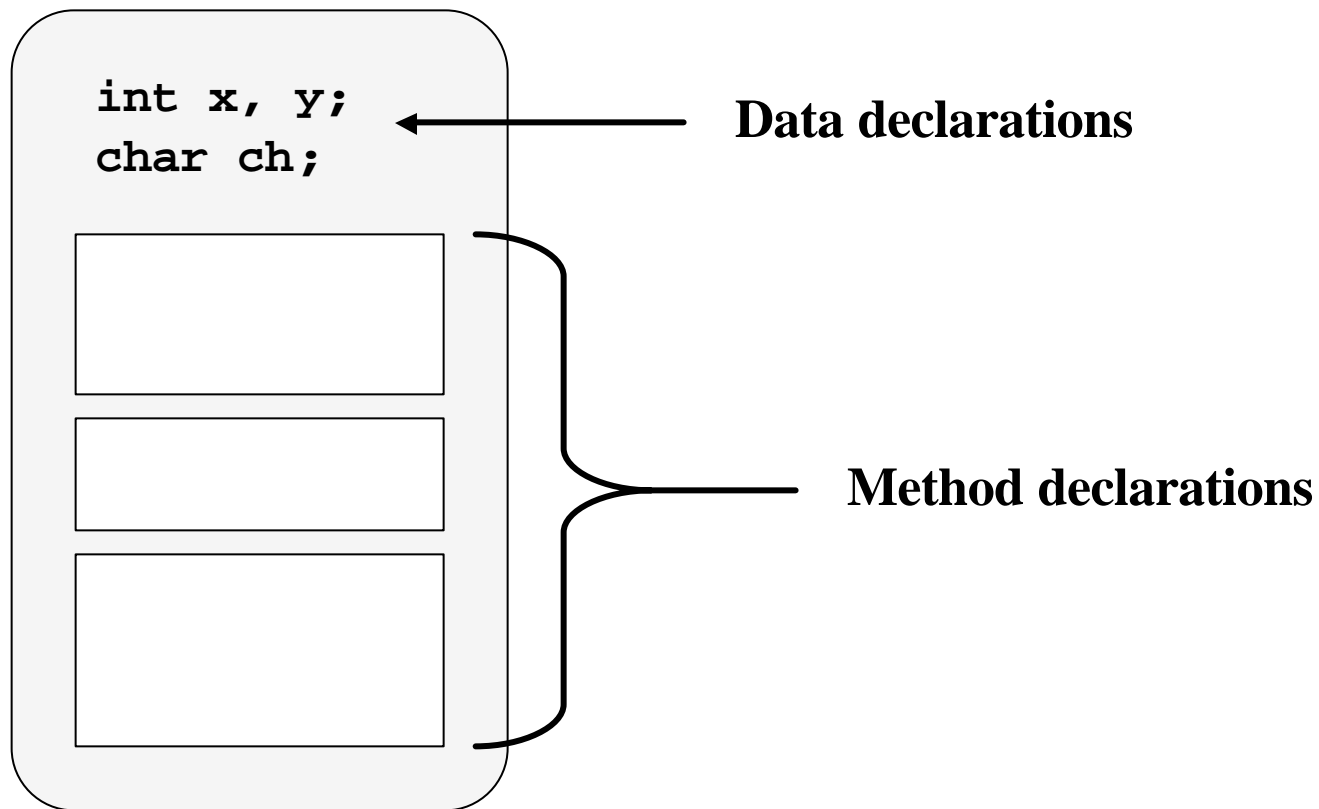
Classes



- The `String` class was provided for us by the Java standard class library
 - But we can also write our own classes that define specific objects that we need
 - For example, suppose we wanted to write a program that simulates the flipping of a coin
 - We could write a `Coin` class to represent a coin object
- 

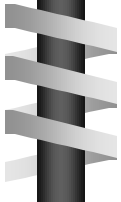
Classes

- A class contains data declarations and method declarations



Data Scope



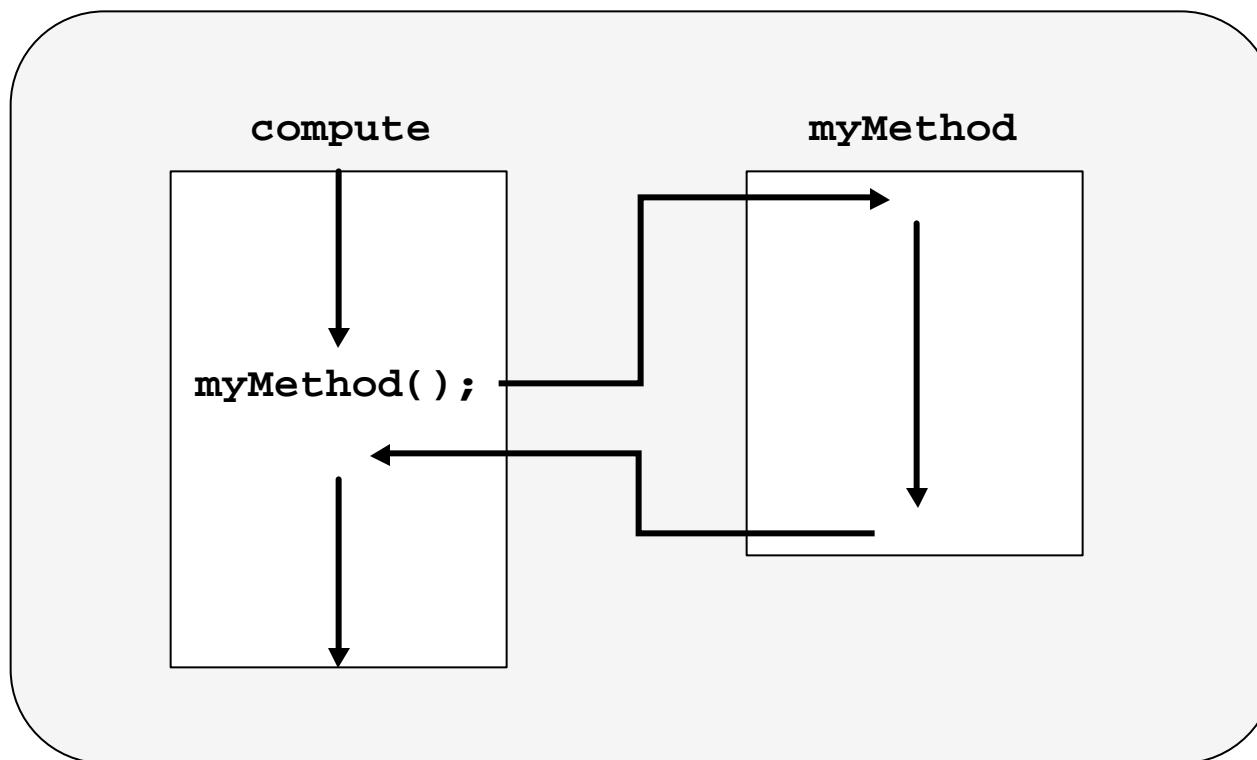
- **The *scope* of data is the area in a program in which that data can be used (referenced)**
 - **Data declared at the class level can be used by all methods in that class**
 - **Data declared within a method can only be used in that method**
 - **Data declared within a method is called *local data***
- 

Writing Methods

- **A *method declaration* specifies the code that will be executed when the method is invoked (or called)**
- **When a method is invoked, the flow of control jumps to the method and executes its code**
- **When complete, the flow returns to the place where the method was called and continues**
- **The invocation may or may not return a value, depending on how the method was defined**

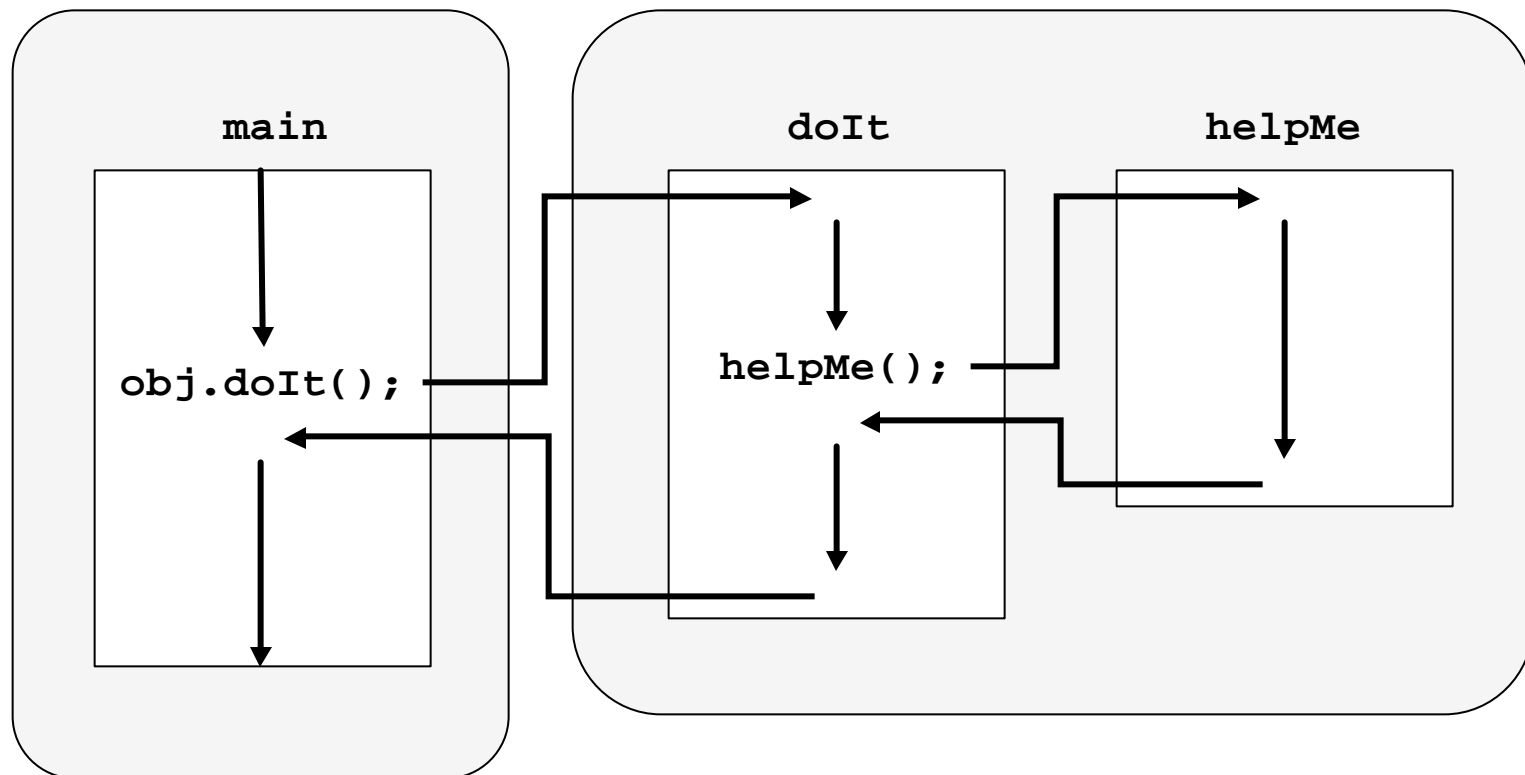
Method Control Flow

- The called method could be within the same class, in which case only the method name is needed



Method Control Flow

- The called method could be part of another class or object



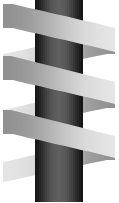
The Coin Class

- ◉ **In our `Coin` class we could define the following data:**
 - `face`, an integer that represents the current face
 - `HEADS` and `TAILS`, integer constants that represent the two possible states

- ◉ **We might also define the following methods:**
 - a `Coin` constructor, to set up the object
 - a `flip` method, to flip the coin
 - a `getFace` method, to return the current face
 - a `toString` method, to return a string description for printing

The Coin Class



- See CountFlips.java (page 179)
 - See Coin.java (page 180)
 - Once the `Coin` class has been defined, we can use it again in other programs as needed
 - Note that the `CountFlips` program did not use the `toString` method
 - A program will not necessarily use every service provided by an object
- 

Instance Data

- The **face** variable in the **Coin** class is called *instance data* because each instance (object) of the **Coin** class has its own
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a **Coin** object is created, a new **face** variable is created as well
- The objects of a class share the method definitions, but they have unique data space
- That's the only way two objects can have different states

Instance Data

- See FlipRace.java (page 182)

class Coin

int face;

coin1

face

0

coin2

face

1

Encapsulation

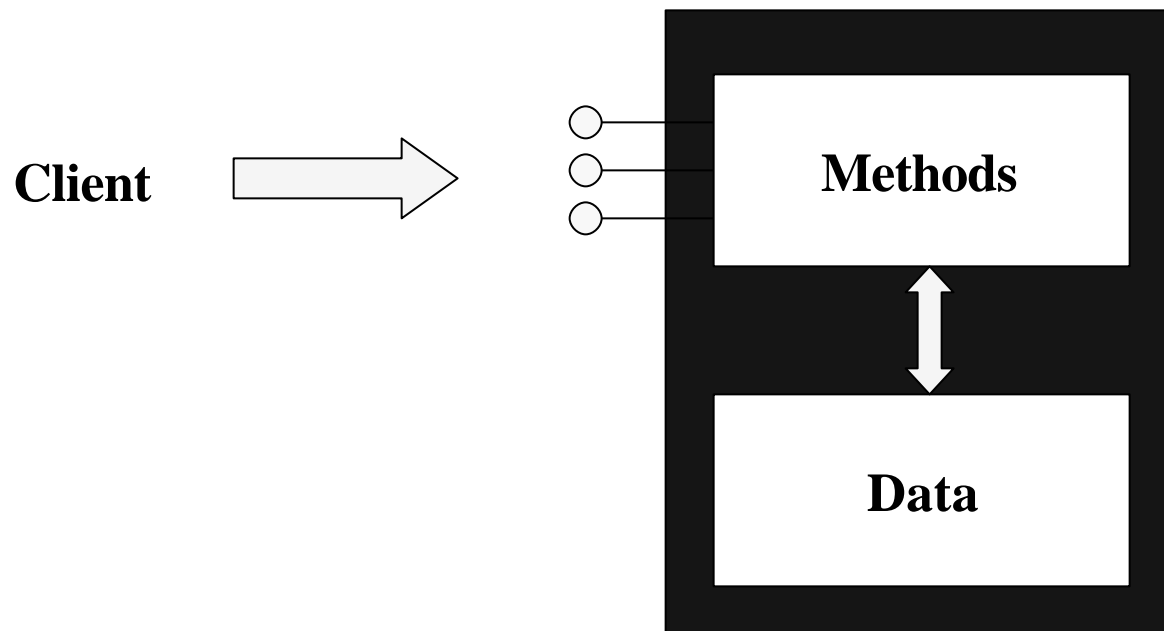
- ⦿ **You can take one of two views of an object:**
 - **internal** - the structure of its data, the algorithms used by its methods
 - **external** - the interaction of the object with other objects in the program
- ⦿ **From the external view, an object is an *encapsulated* entity, providing a set of specific services**
- ⦿ **These services define the *interface* to the object**
- ⦿ **Recall from Chapter 2 that an object is an *abstraction*, hiding details from the rest of the system**

Encapsulation

- ⦿ **An object should be *self-governing***
- ⦿ **Any changes to the object's state (its variables) should be accomplished by that object's methods**
- ⦿ **We should make it difficult, if not impossible, for one object to "reach in" and alter another object's state**
- ⦿ **The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished**

Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which only invokes the interface methods



Visibility Modifiers

- ⦿ In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- ⦿ A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- ⦿ We've used the modifier `final` to define a constant
- ⦿ Java has three visibility modifiers: `public`, `private`, and `protected`
- ⦿ We will discuss the `protected` modifier later

Visibility Modifiers

- ⦿ Members of a class that are declared with *public visibility* can be accessed from anywhere
- ⦿ Members of a class that are declared with *private visibility* can only be accessed from inside the class
- ⦿ Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package
- ⦿ Java modifiers are discussed in detail in Appendix F

Visibility Modifiers

- ⦿ **As a general rule, no object's data should be declared with public visibility**
- ⦿ **Methods that provide the object's services are usually declared with public visibility so that they can be invoked by clients**
- ⦿ **Public methods are also called *service methods***
- ⦿ **A method created simply to assist a service method is called a *support method***
- ⦿ **Since a support method is not intended to be called by a client, it should not be declared with public visibility**

Method Declarations Revisited

- A method declaration begins with a *method header*

`char calc (int num1, int num2, String message)`

The diagram shows the method declaration `char calc (int num1, int num2, String message)`. An arrow points from the text 'return type' to the `char` keyword. Another arrow points from the text 'method name' to the `calc` identifier. A large curly brace spans the entire parameter list `(int num1, int num2, String message)`, with the text 'parameter list' centered below it.

return type

method name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal argument*

Method Declarations

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```

↑
The return expression must be
consistent with the return type

sum and result
are *local data*

They are created each
time the method is called,
and are destroyed when
it finishes executing

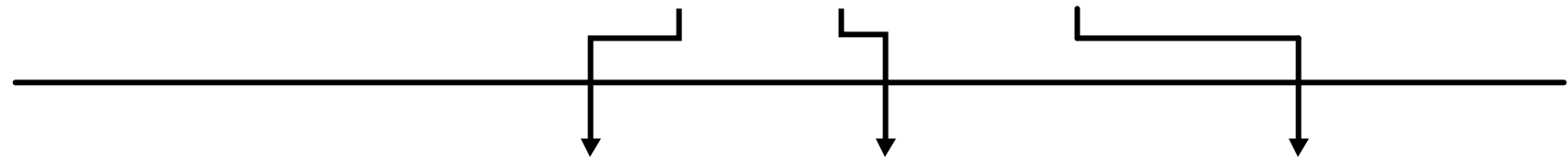
The return Statement

- ⦿ The *return type* of a method indicates the type of value that the method sends back to the calling location
- ⦿ A method that does not return a value has a `void` return type
- ⦿ The *return statement* specifies the value that will be returned
- ⦿ Its expression must conform to the return type

Parameters

- Each time a method is called, the *actual arguments* in the invocation are copied into the formal arguments

```
ch = obj.calc (25, count, "Hello");
```



```
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```


Constructors Revisited

- **Recall that a constructor is a special method that is used to set up a newly created object**
- **When writing a constructor, remember that:**
 - it has the same name as the class
 - it does not return a value
 - it has no return type, not even `void`
 - it often sets the initial values of instance variables
- **The programmer does not have to define a constructor for a class**

Writing Classes

- See BankAccounts.java (page 188)
- See Account.java (page 189)
- An *aggregate object* is an object that contains references to other objects
- An **Account** object is an aggregate object because it contains a reference to a **String** object (that holds the owner's name)
- An aggregate object represents a *has-a relationship*
- A bank account *has a* name

Writing Classes

- ✱ Sometimes an object has to interact with other objects of the same type
- ✱ For example, we might add two `Rational` number objects together as follows:

```
r3 = r1.add(r2);
```

- ✱ One object (`r1`) is executing the method and another (`r2`) is passed as a parameter
- ✱ See `RationalNumbers.java` (page 196)
- ✱ See `Rational.java` (page 197)

Overloading Methods

- ***Method overloading*** is the process of using the same method name for multiple methods
- The ***signature*** of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler must be able to determine which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

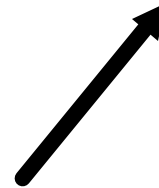
Overloading Methods

Version 1

```
float tryMe (int x)
{
    return x + .375;
}
```

Version 2

```
float tryMe (int x, float y)
{
    return x*y;
}
```



Invocation

```
result = tryMe (25, 4.32)
```

Overloaded Methods

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
etc.
```

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Overloading Methods

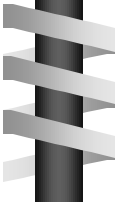
- **Constructors can be overloaded**
- **An overloaded constructor provides multiple ways to set up a new object**
- **See SnakeEyes.java (page 203)**
- **See Die.java (page 204)**

The StringTokenizer Class

- The next example makes use of the `StringTokenizer` class, which is defined in the `java.util` package
- A `StringTokenizer` object separates a string into smaller substrings (tokens)
- By default, the tokenizer separates the string at white space
- The `StringTokenizer` constructor takes the original string to be separated as a parameter
- Each call to the `nextToken` method returns the next token in the string

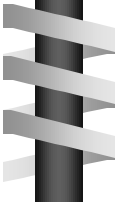
Method Decomposition



- **A method should be relatively small, so that it can be readily understood as a single entity**
 - **A potentially large method should be decomposed into several smaller methods as needed for clarity**
 - **Therefore, a service method of an object may call one or more support methods to accomplish its goal**
 - **See PigLatin.java (page 207)**
 - **See PigLatinTranslator.java (page 208)**
- 

Applet Methods



- In previous examples we've used the `paint` method of the `Applet` class to draw on an applet
 - The `Applet` class has several methods that are invoked automatically at certain points in an applet's life
 - The `init` method, for instance, is executed only once when the applet is initially loaded
 - The `Applet` class also contains other methods that generally assist in applet processing
- 

Graphical Objects

- ⦿ Any object we define by writing a class can have graphical elements
- ⦿ The object must simply obtain a graphics context (a `Graphics` object) in which to draw
- ⦿ An applet can pass its graphics context to another object just as it can any other parameter
- ⦿ See LineUp.java (page 212)
- ⦿ See StickFigure.java (page 215)