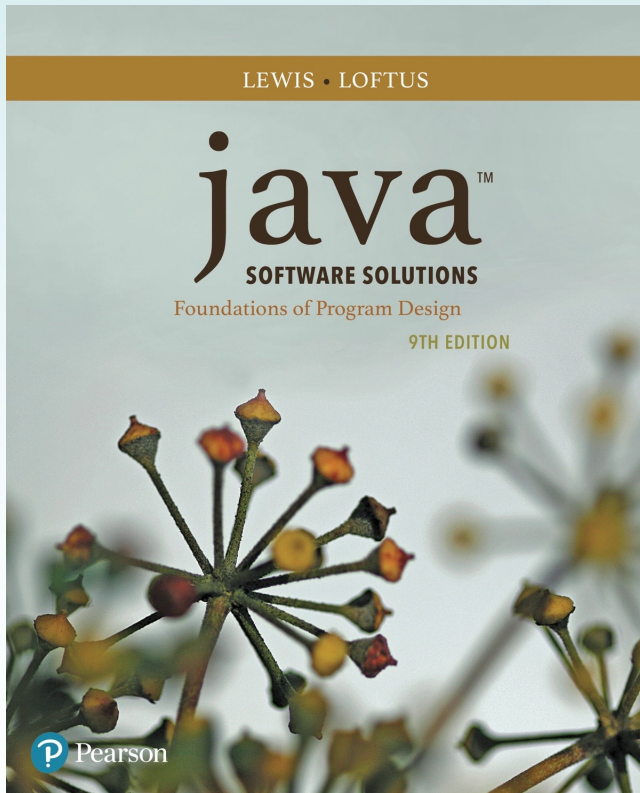


Chapter 4

Writing Classes



Java Software Solutions

Foundations of Program Design

9th Edition

John Lewis
William Loftus

Writing Classes

- We've been using predefined classes from the Java API. Now we will learn to write our own classes.
- Chapter 4 focuses on:
 - class definitions
 - instance data
 - encapsulation and Java modifiers
 - method declaration and parameter passing
 - constructors
 - arcs and images
 - events and event handlers
 - buttons and text fields

Outline



Anatomy of a Class

Encapsulation

Anatomy of a Method

Arcs

Images

Graphical User Interfaces

Text Fields

Writing Classes

- The programs we've written in previous examples have used classes defined in the Java API
- Now we will begin to design programs that rely on classes that we write ourselves
- The class that contains the `main` method is just the starting point of a program
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

Examples of Classes

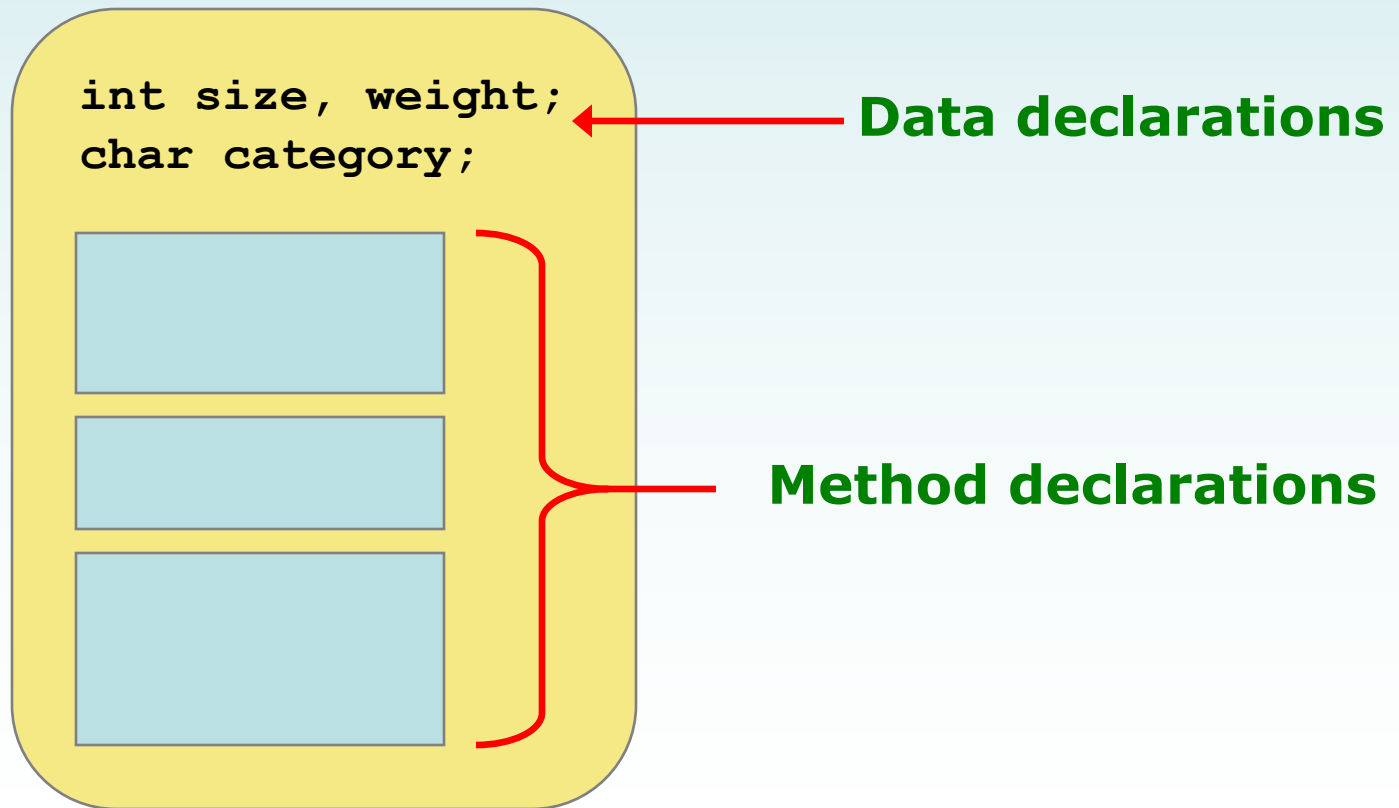
Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Classes and Objects

- Recall from our overview of objects in Chapter 1 that an object has *state* and *behavior*
- Consider a six-sided die (singular of dice)
 - Its state can be defined as which face is showing
 - Its primary behavior is that it can be rolled
- We represent a die by designing a class called `Die` that models this state and behavior
 - The class serves as the blueprint for a die object
- We can then instantiate as many die objects as we need for any particular program

Classes

- A class can contain data declarations and method declarations



Classes

- The values of the data define the state of an object created from the class
- The functionality of the methods define the behaviors of the object
- For our `Die` class, we might declare an integer called `faceValue` that represents the current value showing on the face
- One of the methods would “roll” the die by setting `faceValue` to a random number between one and six

Classes

- We'll want to design the `Die` class so that it is a versatile and reusable resource
- Any given program will probably not use all operations of a given class
- **See** `RollingDice.java`
- **See** `Die.java`

The Die Class

- The `Die` class contains two data values
 - a constant `MAX` that represents the maximum face value
 - an integer `faceValue` that represents the current face value
- The `roll` method uses the `random` method of the `Math` class to determine a new face value
- There are also methods to explicitly set and retrieve the current face value at any time

The toString Method

- It's good practice to define a `toString` method for a class
- The `toString` method returns a character string that represents the object in some way
- It is called automatically when an object is concatenated to a string or when it is passed to the `println` method
- It's also convenient for debugging problems

Constructors

- As mentioned previously, a *constructor* is used to set up an object when it is initially created
- A constructor has the same name as the class
- The `Die` constructor is used to set the initial face value of each new die object to one
- We examine constructors in more detail later in this chapter

Data Scope

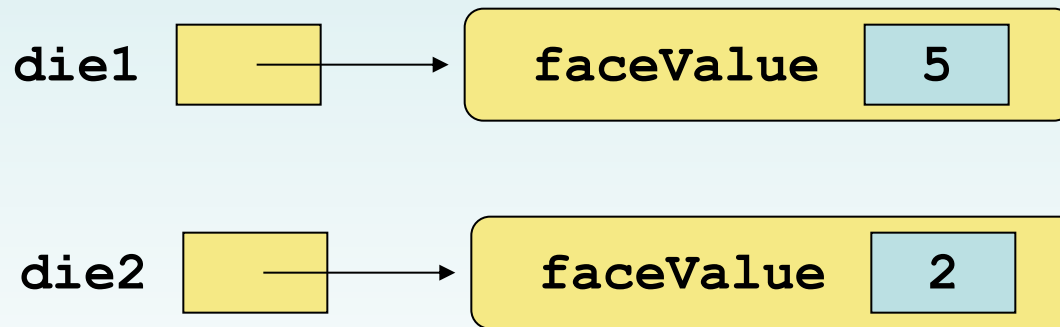
- The *scope* of data is the area in a program in which that data can be referenced (used)
- Data declared at the class level can be referenced by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*
- In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else

Instance Data

- The variable declared at the class level (`faceValue`) is called *instance data*
- Each instance (object) has its own instance variable
- A class declares the type of the data, but it does not reserve memory space for it
- Each time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space
- That's the only way two objects can have different states

Instance Data

- We can depict the two `Die` objects from the `RollingDice` program as follows:



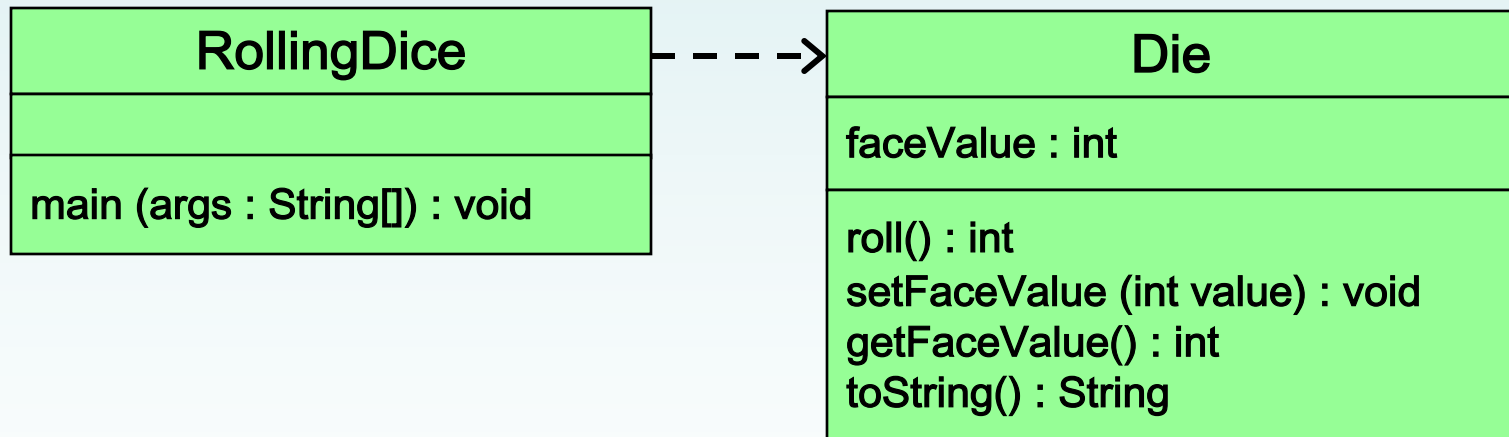
Each object maintains its own `faceValue` variable, and thus its own state

UML Diagrams

- UML stands for the *Unified Modeling Language*
- *UML diagrams* show relationships among classes and objects
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)
- Lines between classes represent *associations*
- A dotted arrow shows that one class *uses* the other (calls its methods)

UML Class Diagrams

- A UML class diagram for the `RollingDice` program:



Quick Check

What is the relationship between a class and an object?

Quick Check

What is the relationship between a class and an object?

A class is the definition/pattern/blueprint of an object. It defines the data that will be managed by an object but doesn't reserve memory space for it. Multiple objects can be created from a class, and each object has its own copy of the instance data.

Quick Check

Where is instance data declared?

What is the scope of instance data?

What is local data?

Quick Check

Where is instance data declared?

At the class level.

What is the scope of instance data?

It can be referenced in any method of the class.

What is local data?

Local data is declared within a method, and is only accessible in that method.

Outline

Anatomy of a Class



Encapsulation

Anatomy of a Method

Arcs

Images

Graphical User Interfaces

Text Fields

Encapsulation

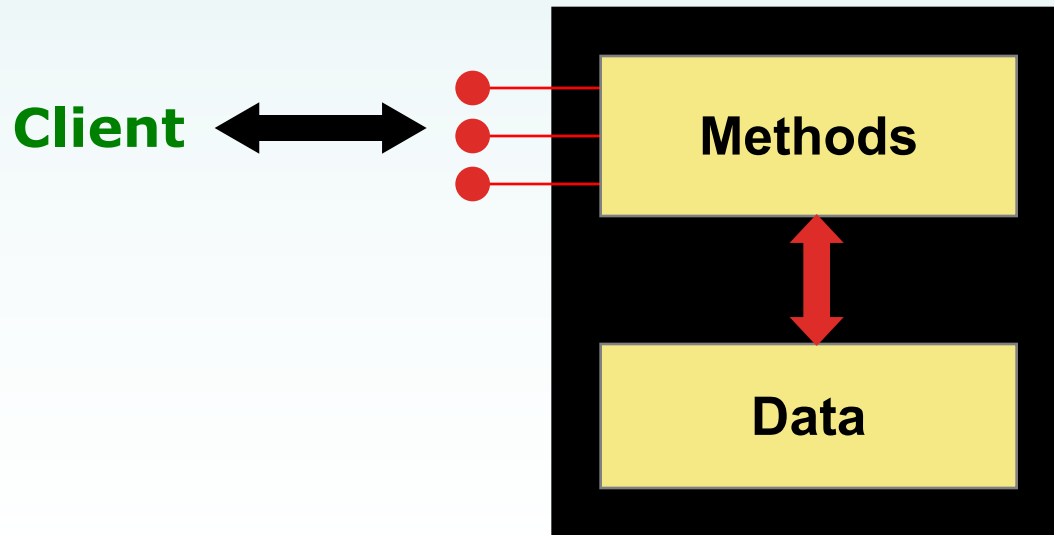
- There are two views of an object:
 - internal - the details of the variables and methods of the class that defines it
 - external - the services that an object provides and how the object interacts with the rest of the system
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object

Encapsulation

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made by that object's methods
- We should make it difficult, if not impossible, for a client to access an object's variables directly
- That is, an object should be *self-governing*

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods and they manage the instance data



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
- We've used the `final` modifier to define constants
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility Modifiers

- Members of a class that are declared with *public visibility* can be referenced anywhere
- Members of a class that are declared with *private visibility* can be referenced only within that class
- Members declared without a visibility modifier have *default visibility* and can be referenced by any class in the same package
- An overview of all Java modifiers is presented in Appendix E

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to modify the values directly
- Therefore instance variables should not be declared with public visibility
- It is acceptable to give a constant public visibility, which allows it to be used outside of the class
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed

Visibility Modifiers

- Methods that provide the object's services are 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

Visibility Modifiers

	<code>public</code>	<code>private</code>
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values
- An *accessor method* returns the current value of a variable
- A *mutator method* changes the value of a variable
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where X is the name of the value
- They are sometimes called “getters” and “setters”

Mutator Restrictions

- The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state
- A mutator is often designed so that the values of variables can be set only within particular limits
- For example, the `setFaceValue` mutator of the `Die` class should restrict the value to the valid range (1 to `MAX`)
- We'll see in Chapter 5 how such restrictions can be implemented

Quick Check

Why was the `faceValue` variable declared as `private` in the `Die` class?

Why is it ok to declare `MAX` as `public` in the `Die` class?

Quick Check

Why was the `faceValue` variable declared as `private` in the `Die` class?

By making it `private`, each `Die` object controls its own data and allows it to be modified only by the well-defined operations it provides.

Why is it ok to declare `MAX` as `public` in the `Die` class?

`MAX` is a constant. Its value cannot be changed. Therefore, there is no violation of encapsulation.

Outline

Anatomy of a Class

Encapsulation



Anatomy of a Method

Arcs

Images

Graphical User Interfaces

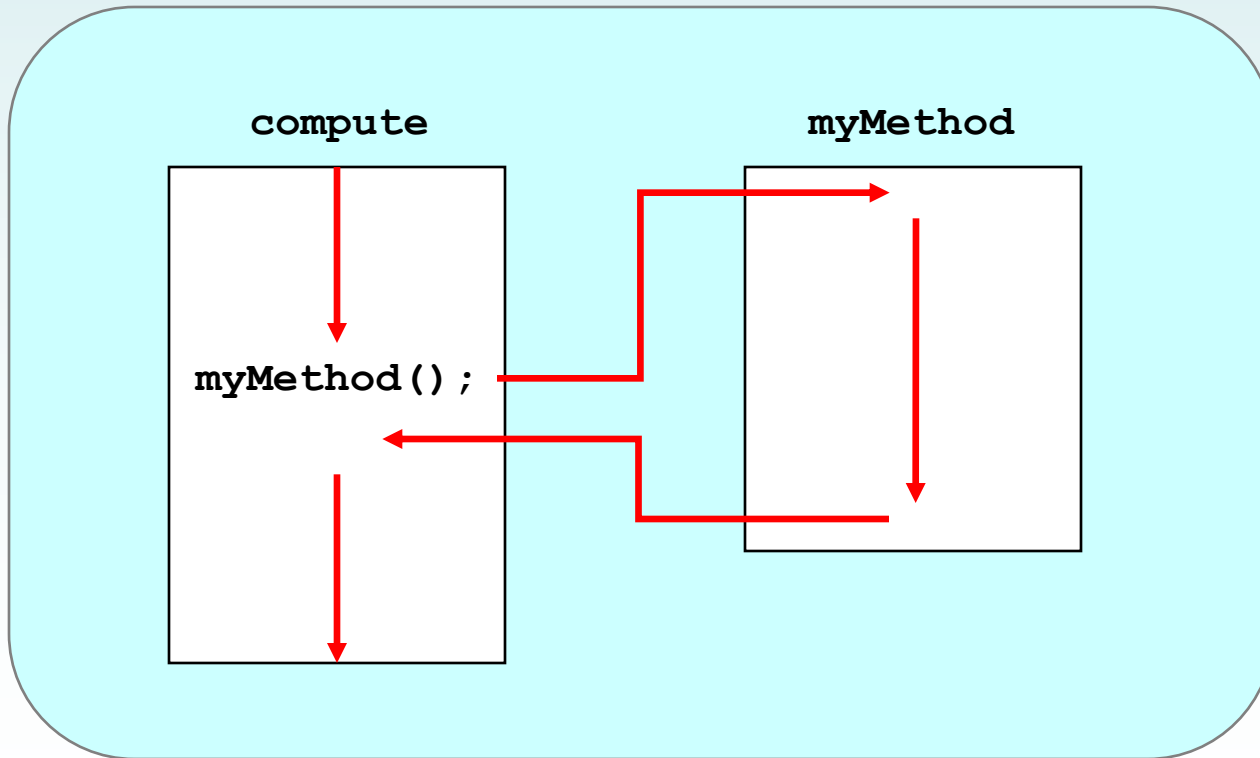
Text Fields

Method Declarations

- Let's now examine methods in more detail
- A *method declaration* specifies the code that will be executed when the method is invoked (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 is defined

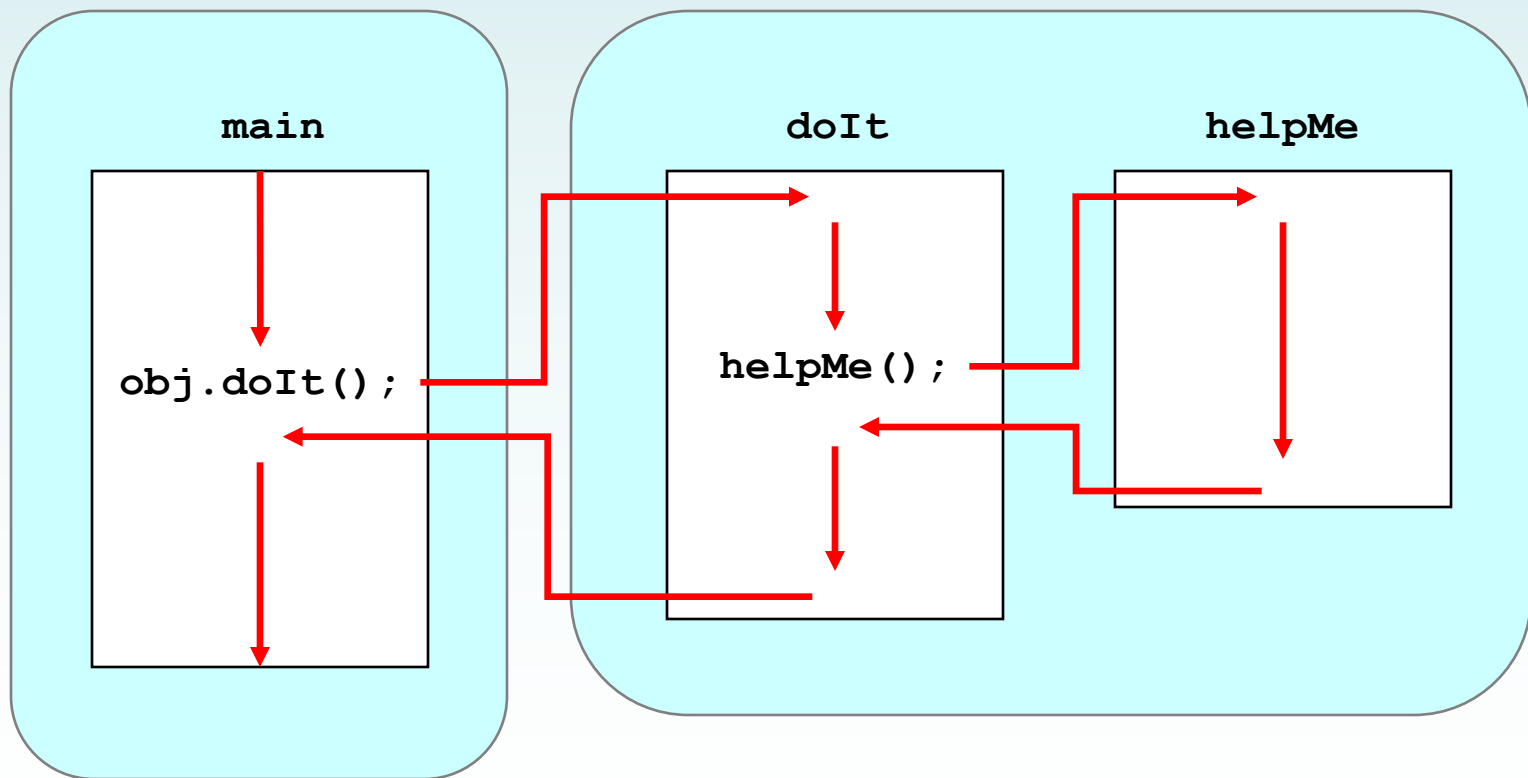
Method Control Flow

- If the called method is in the same class, only the method name is needed



Method Control Flow

- The called method is often part of another class or object



Method Header

- A method declaration begins with a *method header*

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

The diagram shows the method header `char calc(int num1, int num2, String message)` with three red arrows pointing to its components: one to `char` (labeled **return type**), one to `calc` (labeled **method name**), and one to the opening parenthesis (labeled **parameter list**). A red curly brace is positioned below the parameter list, spanning from the opening parenthesis to the closing parenthesis.

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 parameter*

Method Body

- 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
- A *return statement* specifies the value that will be returned


`return expression;`

- Its expression must conform to the return type

Parameters

- When a method is called, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header

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



A diagram illustrating the process of passing actual parameters to formal parameters. A horizontal green line separates the method invocation from the method definition. Three red arrows originate from the arguments in the invocation: the first arrow starts under '25' and points to 'int num1'; the second arrow starts under 'count' and points to 'int num2'; the third arrow starts under '"Hello"' and points to 'String message'.

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

    return result;
}
```

Local Data

- As we've seen, local variables can be declared inside a method
- The formal parameters of a method create *automatic local variables* when the method is invoked
- When the method finishes, all local variables are destroyed (including the formal parameters)
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists

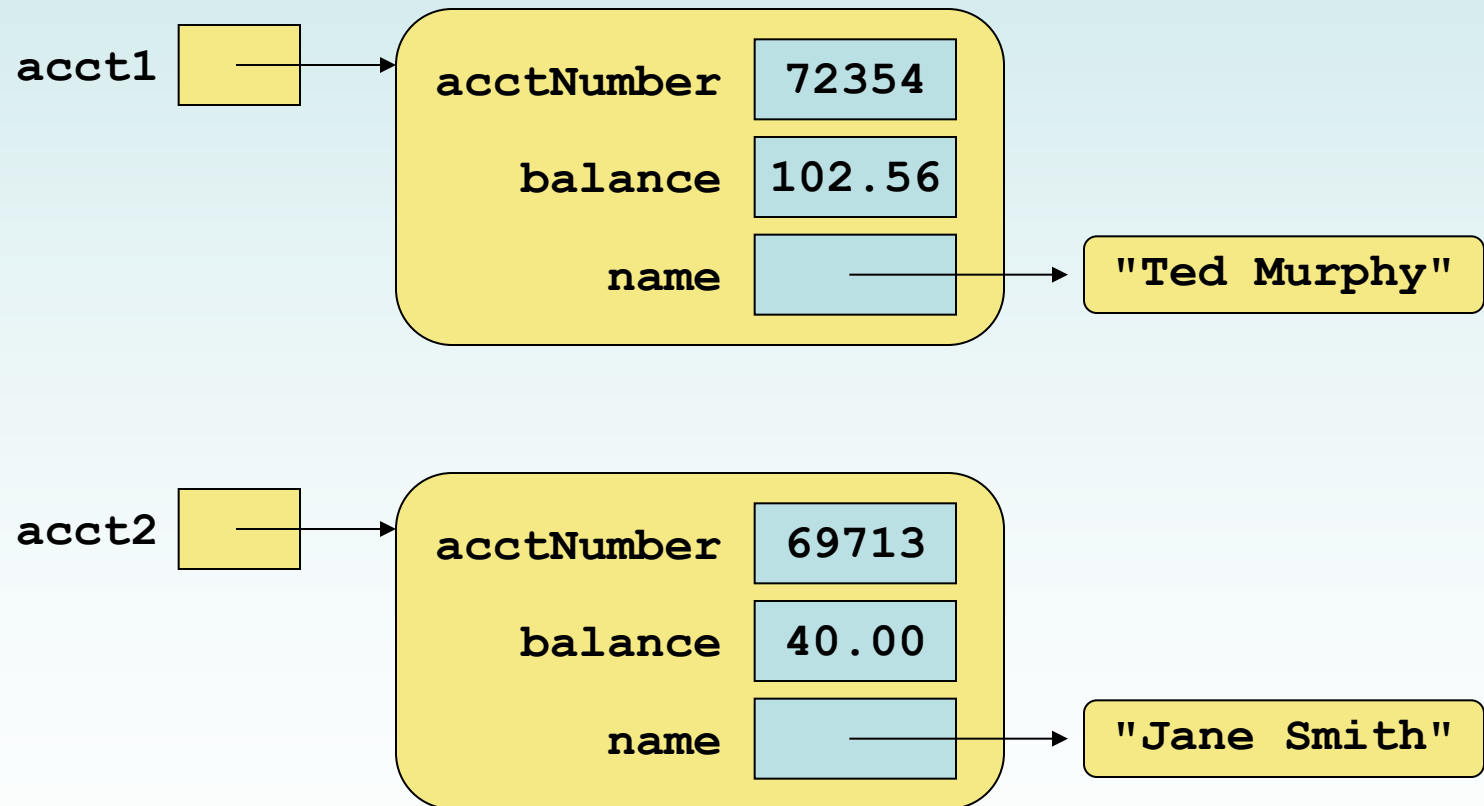
Bank Account Example

- Let's look at another example that demonstrates the implementation details of classes and methods
- We'll represent a bank account by a class named `Account`
- It's state can include the account number, the current balance, and the name of the owner
- An account's behaviors (or services) include deposits and withdrawals, and adding interest

Driver Programs

- A *driver program* drives the use of other, more interesting parts of a program
- Driver programs are often used to test other parts of the software
- The `Transactions` class contains a `main` method that drives the use of the `Account` class, exercising its services
- See `Transactions.java`
- See `Account.java`

Bank Account Example



Bank Account Example

- There are some improvements that can be made to the `Account` class
- Formal getters and setters could have been defined for all data
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw` method is positive

Constructors Revisited

- Note that a constructor has no return type specified in the method header, not even `void`
- A common error is to put a return type on a constructor, which makes it a “regular” method that happens to have the same name as the class
- The programmer does not have to define a constructor for a class
- If no constructors are explicitly defined in a class, a *default constructor* is created that has no parameters

Quick Check

How do we express which `Account` object's balance is updated when a deposit is made?

Quick Check

How do we express which `Account` object's balance is updated when a deposit is made?

Each account is referenced by an object reference variable:

```
Account myAcct = new Account (...);
```

and when a method is called, you call it through a particular object:

```
myAcct.deposit(50);
```

Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method



Arcs

Images

Graphical User Interfaces

Text Fields

Arcs

- In Chapter 3 we explored basic shapes: lines, rectangles, circles, and ellipses
- In JavaFX, an arc is defined as a portion of an ellipse
- Like an ellipse, the first four parameters to the `Arc` constructor specify the center point (x and y) as well as the radii along the horizontal and vertical
- Two additional parameters specify the portion of the ellipse that define the arc

Arcs

- The Arc constructor:

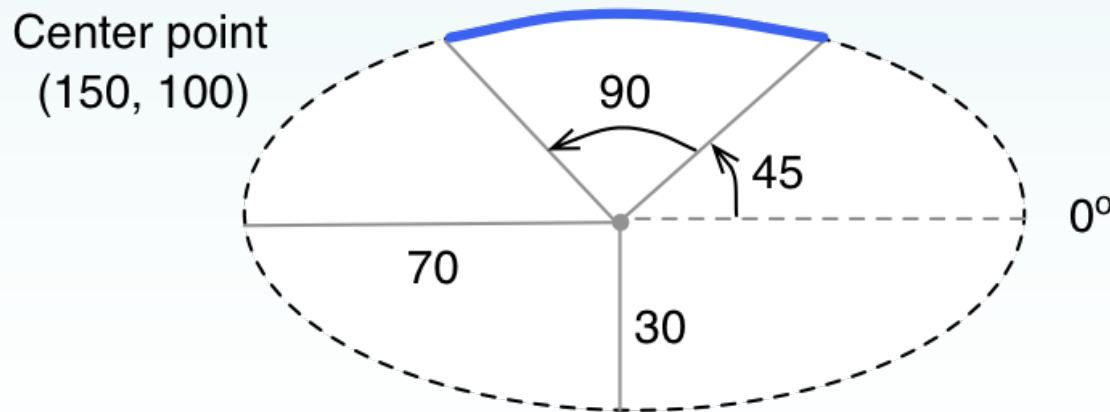
```
Arc(centerX, centerY, radiusX, radiusY,  
    startAngle, arcLength)
```

- The *start angle* is where the arc begins relative to the horizontal
- The *arc length* is the angle that defines how big the arc is
- Both angles are specified in degrees

Arcs

- An arc whose underlying ellipse is centered at (150, 100), a horizontal radius of 70 and a vertical radius of 30, a start angle of 45 and a arc length of 90:

```
Arc myArc = new Arc(150, 100, 70, 30,  
                    45, 90);
```



Arcs

- An arc also has an *arc type*:

ArcType.OPEN	The curve along the ellipse edge
ArcType.CHORD	End points are connected by a straight line
ArcType.ROUND	End points are connected to the center point of the ellipse, forming a rounded “pie” piece

- See `ArcDisplay.java`

Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method

Arcs



Images

Graphical User Interfaces

Text Fields

Images

- The JavaFX `Image` class is used to load an image from a file or URL
- Supported formats: jpeg, gif, and png
- To display an image, use an `ImageView` object
- An `Image` object cannot be added to a container directly
- See `ImageDisplay.java`

Layout Panes

- This example uses a `StackPane` instead of a `Group` as the root node of the scene
- A stack pane is a JavaFX *layout pane* (one of several), which governs how its contents are presented
- A stack pane stacks its nodes on top of each other
- Since the image view is the only node in the pane, the stack pane simply serves to keep the image centered in the window

Layout Panes

- The background color of a layout pane is set using a call to the `setStyle` method
- The `setStyle` method accepts a string that can specify various style properties
- The notation used for JavaFX style properties are similar to cascading style sheets (CSS), used to specify the look of HTML elements on a Web page
- JavaFX style property names begin with the prefix “-fx-”

Images

- The parameter to the `Image` constructor can include a pathname:

```
Image logo = new Image("myPix/smallLogo.png");
```

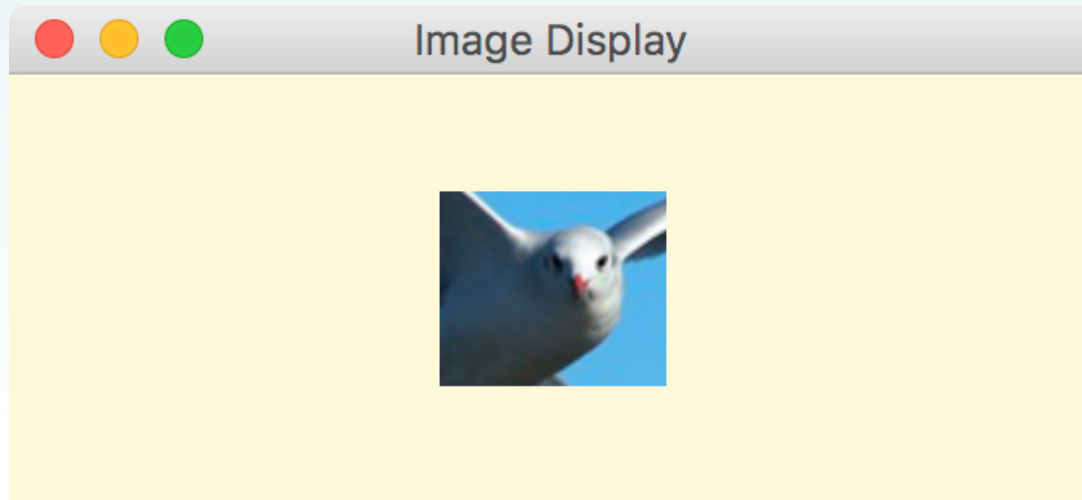
- It can also be a URL:

```
Image logo = new Image("http://example.com/images/bio.jpg");
```

Viewports

- A *viewport* is a rectangular area that restricts the pixels displayed in an `ImageView`
- It is defined by a `Rectangle2D` object:

```
imgView.setViewport(new Rectangle2D(200, 80, 70, 60));
```



Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method

Arcs

Images



Graphical User Interfaces

Text Fields

Graphical User Interfaces

- A Graphical User Interface (GUI) in Java is created with at least three kinds of objects:
 - controls, events, and event handlers
- A *control* is a screen element that displays information or allows the user to interact with the program:
 - labels, buttons, text fields, sliders, etc.

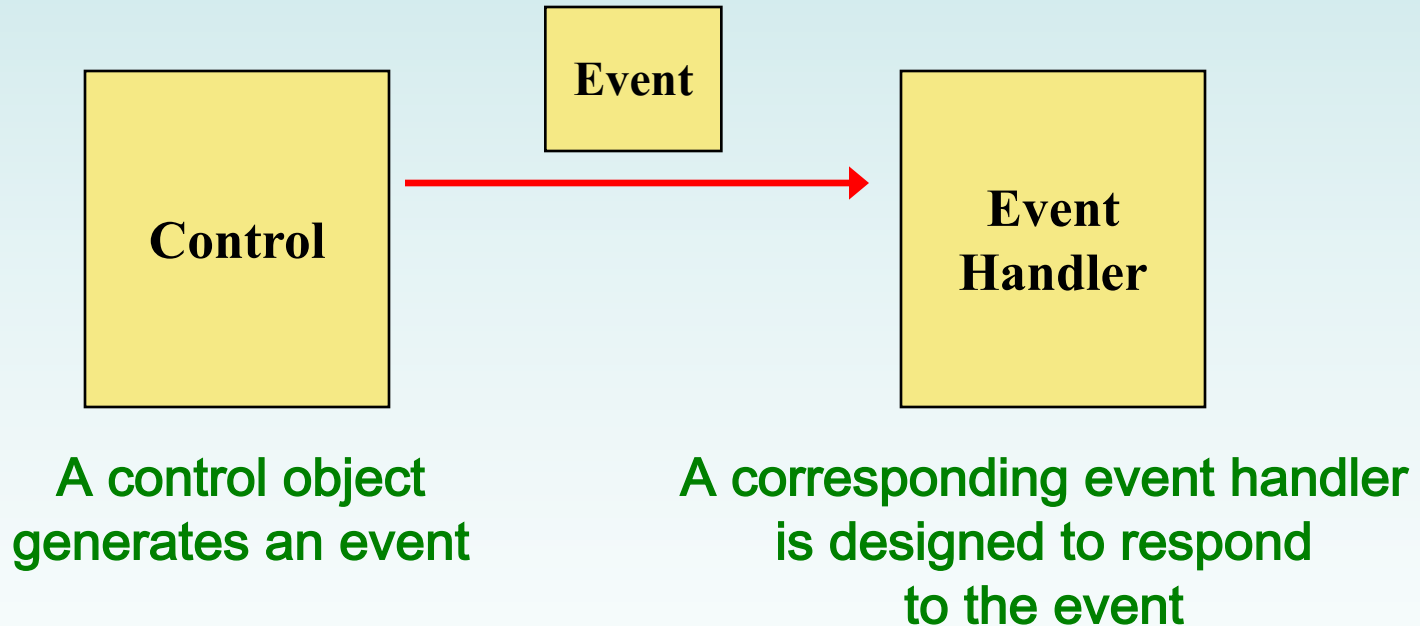
Graphical User Interfaces

- An *event* is an object that represents some activity to which we may want to respond
- For example, we may want our program to perform some action when the following occurs:
 - a graphical button is pressed
 - a slider is dragged
 - the mouse is moved
 - the mouse is dragged
 - the mouse button is clicked
 - a keyboard key is pressed

Graphical User Interfaces

- The Java API contains several classes that represent typical events
- Controls, such as a button, generate (or fire) an event when it occurs
- We set up an *event handler* object to respond to an event when it occurs
- We design event handlers to take whatever actions are appropriate when an event occurs

Graphical User Interfaces



When the event occurs, the control calls the appropriate method of the listener, passing an object that describes the event

Graphical User Interfaces

- A JavaFX *button* is defined by the `Button` class
- It generates an *action event*
- The `PushCounter` example displays a button that increments a counter each time it is pushed
- See `PushCounter.java`

Graphical User Interfaces

- A call to the `setOnAction` method sets up the relationship between the button that generates the event and the event handler that responds to it
- This example uses a *method reference* (using the `::` operator) to specify the event handler method
- The `this` reference indicates that the event handler method is in the same class
- So the `PushCounter` class also represents the event handler for this program

Graphical User Interfaces

- The event handler method can be called whatever you want, but must accept an `ActionEvent` object as a parameter
- In this example, the event handler method increments the counter and updates the text object
- The counter and `Text` object are declared at the class level so that both methods can use them

Graphical User Interfaces

- In this example, a `FlowPane` is used as the root node of the scene
- A flow pane is another layout pane, which displays its contents horizontally in rows or vertically in columns
- A gap of 20 pixels is established between elements on a row using the `setHGap` method

Alternate Event Handlers

- Instead of using a method reference, the event handler could be specified using a separate class that implements the `EventHandler` interface:

```
public class ButtonHandler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent event)
    {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```

Alternate Event Handlers

- The event handler class could be defined as public in a separate file or as a private inner class in the same file
- Either way, the call to the `setOnAction` method would specify a new event handler object:

```
push.setOnAction(new ButtonHandler());
```


Alternate Event Handlers

- Another approach would be to define the event handler using a *lambda expression* in the call to `setOnAction`:

```
push.setOnAction( (event) -> {  
    count++;  
    countText.setText("Pushes: " + count);  
} ) ;
```

- A lambda expression is defined by a set of parameters, the `->` operator and an expression

Alternate Event Handlers

- A lambda expression can be used whenever an object of a *functional interface* is required
- A functional interface contains a single method
- The `EventHandler` interface is a functional interface
- The method reference approach is equivalent to a lambda expression

Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method

Arcs

Images

Graphical User Interfaces



Text Fields

Text Fields

- Let's look at a GUI example that uses another type of control
- A *text field* allows the user to enter one line of input
- If the cursor is in the text field, the text field object generates an action event when the enter key is pressed
- See `FahrenheitConverter.java`
- See `FahrenheitPane.java`

Text Fields

- The details of the user interface are set up in a separate class that extends `GridPane`
- `GridPane` is a JavaFX layout pane that displays nodes in a rectangular grid
- The GUI elements are set up in the constructor of `FahrenheitPane`
- The event handler method is also defined in `FahrenheitPane`

Text Fields

- Through inheritance, a `FahrenheitPane` is a `GridPane` and inherits the `add` method
- The parameters to `add` specify the grid cell to which to add the node
- Row and column numbering in a grid pane start at 0
- When the user presses return, the event handler method is called, which converts the value and updates the text result

Summary

- Chapter 4 focused on:
 - class definitions
 - instance data
 - encapsulation and Java modifiers
 - method declaration and parameter passing
 - constructors
 - arcs and images
 - events and event handlers
 - buttons and text fields