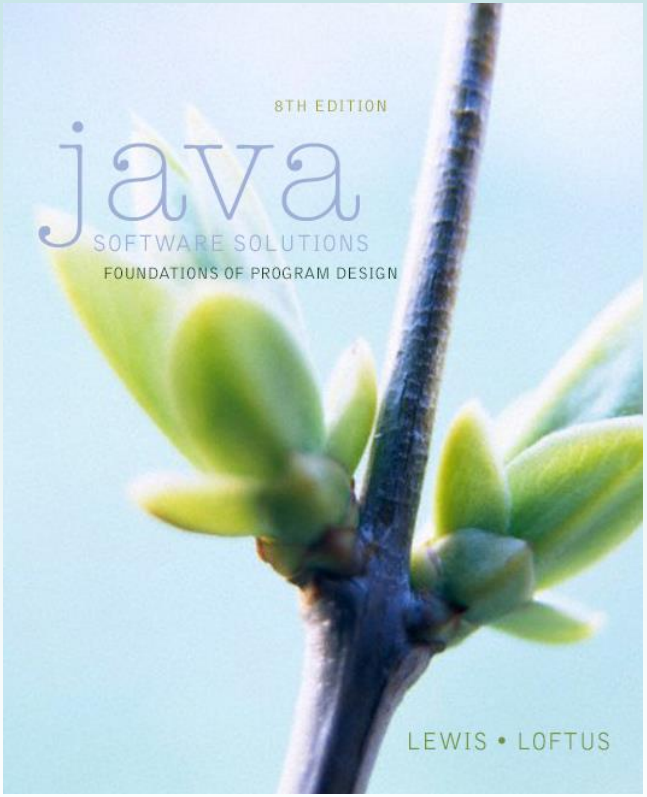
Chapter 4
Writing Classes



Java Software Solutions

Foundations of Program Design

8th Edition

revised 01/10/2017

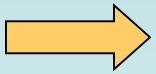
John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

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
  - ☺ graphical objects
  - ☺ events and listeners
  - ☺ buttons and text fields

Outline



Anatomy of a Class

Encapsulation

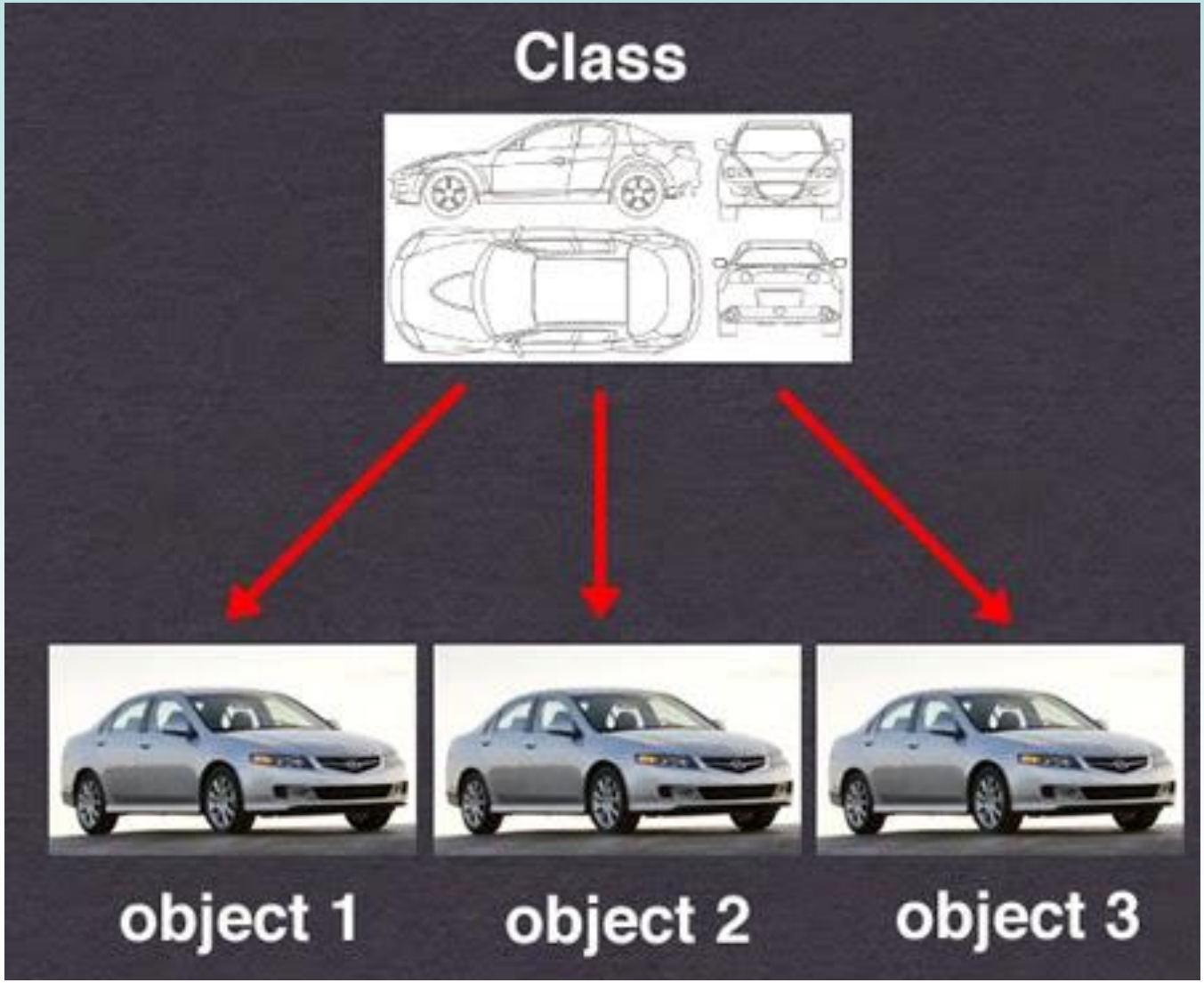
Anatomy of a Method

☺ **Graphical Objects**

☺ **Graphical User Interfaces**

☺ **Buttons and Text Fields**

Classes and Objects



Writing Classes

- Previous examples have used classes from the Java standard class library
- Now we will write classes ourselves
- The class that contains the `main` method is the starting point of a program
- Object-oriented programming is defining classes that represent objects with well-defined characteristics and functionality

Examples of Classes

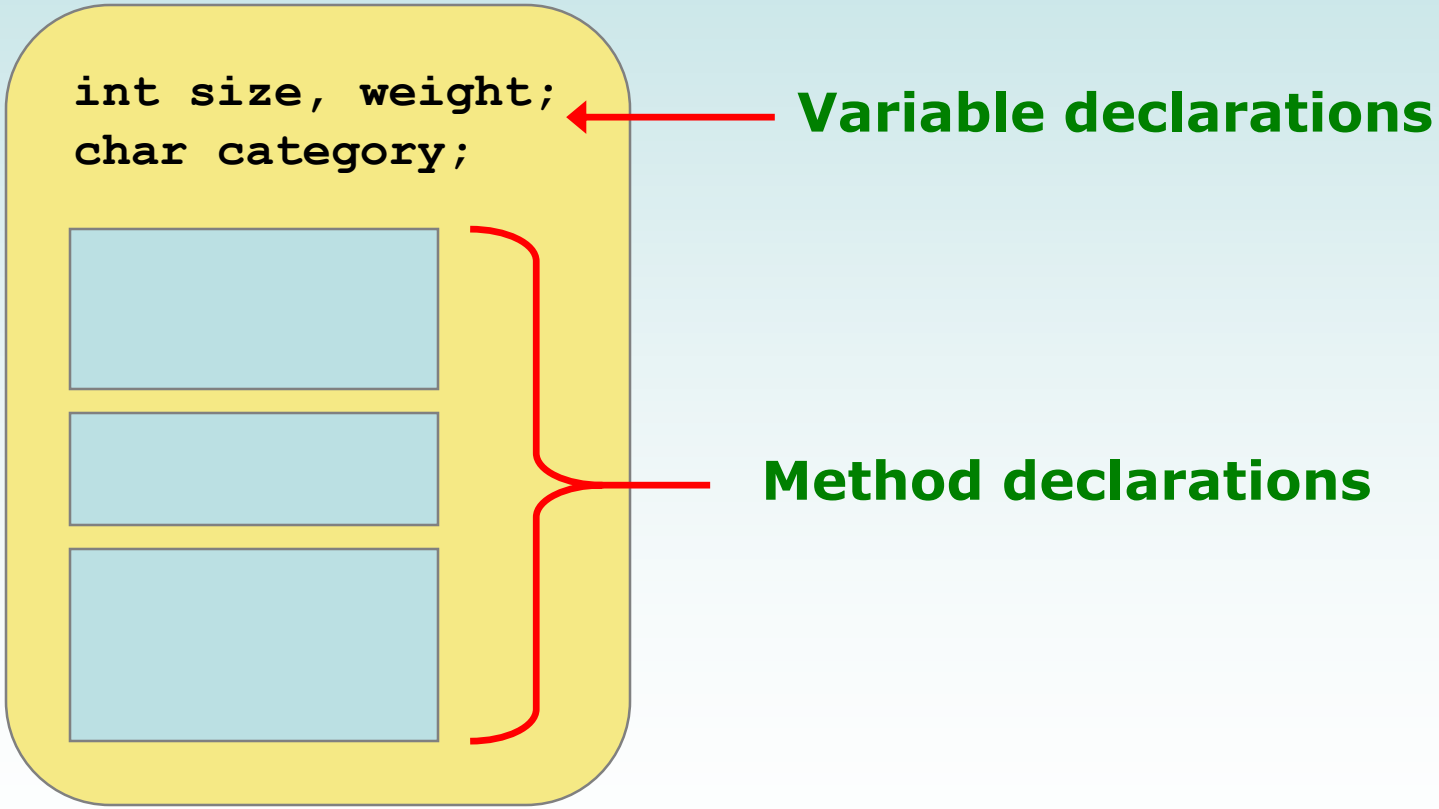
Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
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: an object has *state* and *behavior*
- Consider a six-sided die:
  - It's state is which face is showing
  - It's behavior is that it can be rolled
- Represent a die with a class called `Die` that models this state and behavior
  - The class is a 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 variable declarations (state) and method declarations (behavior)



Classes

- The values of its variables define the state of an object
- The methods define the behaviors of the object
- For our `Die` class, an integer called `faceValue` holds the current face
- A method “rolls” the die by setting `faceValue` to a random number between one and six

Classes

- Design the `Die` class so that it is versatile and reusable
- See `RollingDice.java`
- See `Die.java`

```java
//********************************************************************
//  RollingDice.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of a user-defined class.
//********************************************************************

public class RollingDice
{
   //-----------------------------------------------------------------
   //  Creates two Die objects and rolls them several times.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Die die1, die2;
      int sum;

      die1 = new Die();
      die2 = new Die();

      die1.roll();   // roll the die and change its state
      die2.roll();

      System.out.println("Die One: " + die1 + ", Die Two: " + die2);
```

```
        die1.roll();
        die2.setFaceValue(4);
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);

        sum = die1.getFaceValue() + die2.getFaceValue();
        System.out.println("Sum: " + sum);

        sum = die1.roll() + die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        System.out.println("New sum: " + sum);
    }
}
```

`"Die One: " + die1`

— this is string concatenation. The object reference die1 is used to find an object. That object creates a string using its `toString()` method, which is concatenated to the `"Die One: "` string.

```
        die1.roll();
        die2.setFaceValue(4);
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);

        sum = die1.getFaceValue() + die2.getFaceValue();
        System.out.println("Sum: " + sum);

        sum = die1.roll() + die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        System.out.println("New sum: " + sum);
    }
}
```

Sample Run

```
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6
```

```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;      // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

```
//-----------------------------------------------------------------
//  Rolls the die and returns the result.
//-----------------------------------------------------------------
public int roll()
{
   faceValue = (int)(Math.random() * MAX) + 1;
   return faceValue;
}

//-----------------------------------------------------------------
//  Face value mutator.
//-----------------------------------------------------------------
public void setFaceValue(int value)
{
   faceValue = value;
}

//-----------------------------------------------------------------
//  Face value accessor.
//-----------------------------------------------------------------
public int getFaceValue()
{
   return faceValue;
}
```

```
    //-----------------------------------------------------------------
    //  Returns a string representation of this die.
    //  Do this by concatenation with the empty string.
    //-----------------------------------------------------------------
    public String toString()
    {
       String result =  "" + faceValue ;

       return result;
    }
}
```

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
  - could use a Random object, but does this instead
- There are also methods to explicitly set and retrieve the current face value

The `toString` Method

- It's good practice to define a `toString` method for every class
- The `toString` method returns a reference to a String that represents the object
- All classes automatically have a `toString` method, but it might not do what you want
- 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 to use for debugging

Constructors

- A *constructor* sets up an object when it is created
- A constructor has the same name as the class
- The `Die` constructor sets the initial face value of each new die object to one
- A great deal of work is done by the run-time system to create an object
  - all that your constructor does is put in some “final touches”

Scope

- The *scope* of a variable is the area in a source program in which that variable can be referenced (used)
- A variable declared at the class level (an *instance variable*) can be used by all methods in that class
  - instance variables hold their values as long as their object exists
  - their value may be changed with an assignment statement
- A variable declared within a method can be used only in that method
  - A variable declared within a method is called a *local variable*
- In the `Die` class, the variable `result` is declared inside the `toString` method — it is local to that method and cannot be used anywhere else

Instance Variables

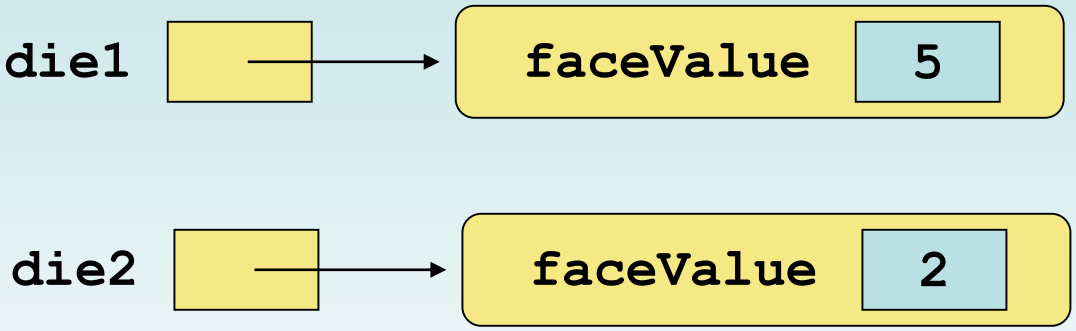
- A variable declared at the class level (such as `faceValue`) is called an *instance variable*
- Each instance (object) has its own instance variables
- A class describes (declares) the type of the data, but it does not reserve memory space for it
  - that happens when an object is constructed
- Each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

Instance Data

- The objects of a class share method definitions, but each object has its own data
- Conceptually, each object has its own methods, identical to the methods of every other object of the same class.
  - In reality, objects don’t need their own copy and so share the actual code with other objects of the same type.

Instance Data

- The two `Die` objects from the `RollingDice` program:



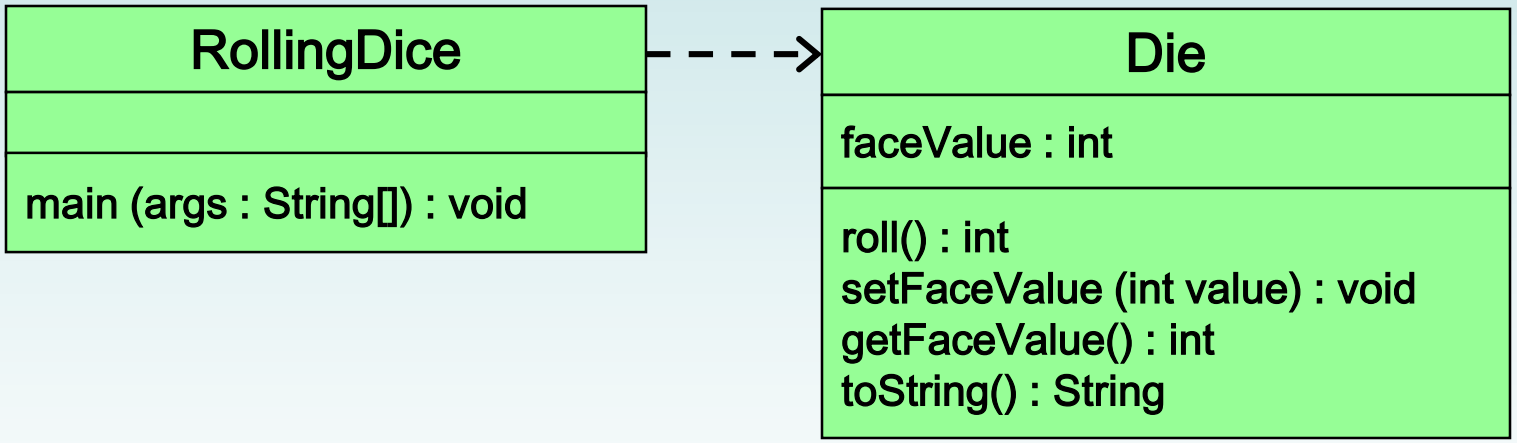
Each object maintains its own `faceValue` variable, and thus its own state

UML Diagrams

- UML stands for the ***U****nified* ***M****odeling* ***L****anguage*
- *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, state (data), and behavior (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 for an object. It defines the data that will be managed by an object but doesn't reserve memory space for it.

Many objects can be created from a class, and each object has its own instance data.

Quick Check

Where are instance variables declared?

What is the scope of instance variables?

What are local variables?

Quick Check

Where are instance variables declared?

At the class level.

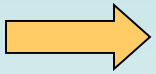
What is the scope of instance variables?

They can be referenced in any method of the class.

What are local variables?

Local variables are declared within a method, and are only accessible in that method.

Outline



Anatomy of a Class

Encapsulation

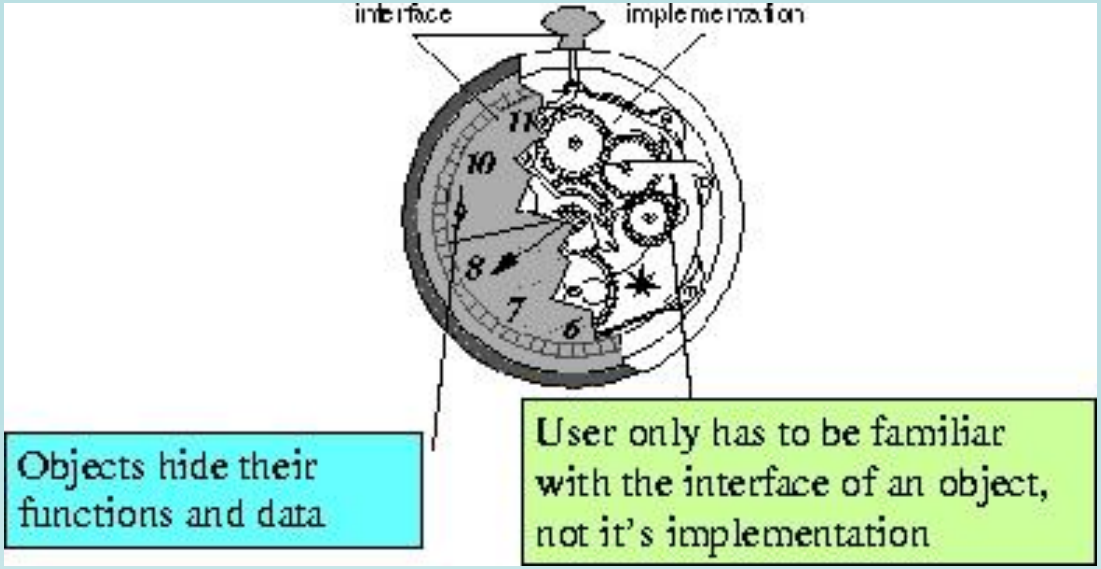
Anatomy of a Method

Graphical Objects

Graphical User Interfaces

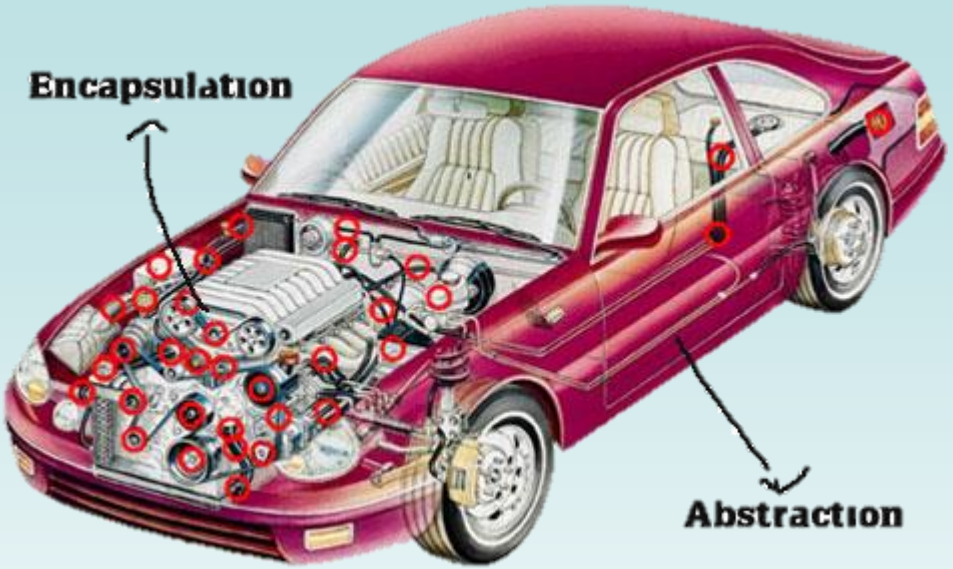
Buttons and Text Fields

Encapsulation



- 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

Encapsulation



- 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

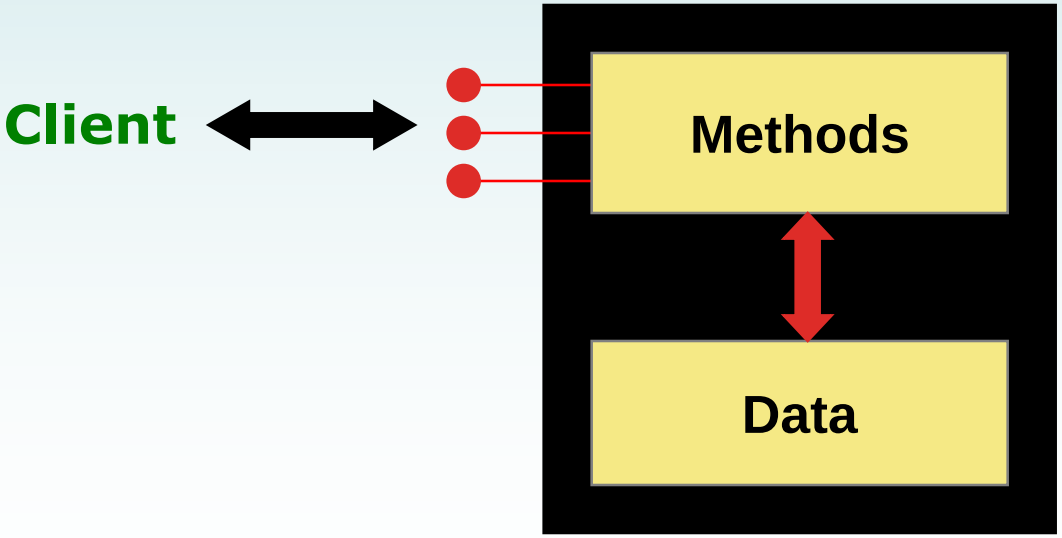
- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods) without knowing how those services are accomplished
- Any changes to an object's state (its variables) should be made only by that object's methods
- 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

- One object (called the *client*) may use another object for the services it provides
  - We have used methods of String objects without knowing the details of how they work.
  - You can use a car object without knowing the details of how it works.
  - Other cars should not be able to affect the inner workings of your car.
- 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

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, encapsulation is done using *visibility modifiers*
- A *modifier* is a word that specifies the characteristics of a method or data
- The `final` modifier defines constants
- 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
  - "member" means instance variable or method
- 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

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to modify the values directly
- Instance variables should be declared with private 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 them, their values 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 be declared with private visibility

Visibility Modifiers

	`public`	`private`
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

Accessors and Mutators

- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value
  - this is a convention among programmers, not a requirement of syntax
- They are sometimes called *getters* and *setters*

Mutator Restrictions

- The use of mutators restricts a client's options to modify an object's state
  - enforces modularity
  - you don't want your car stereo to affect the brakes
- A mutator often checks that values are OK
- For example, the `setFaceValue` mutator of the `Die` class should restrict the value to the valid range (`1` to `MAX`)

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.

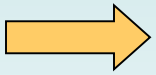
Why is it ok to declare `MAX` as `public` in the `Die` class?

`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

☺ **Graphical Objects**

☺ **Graphical User Interfaces**

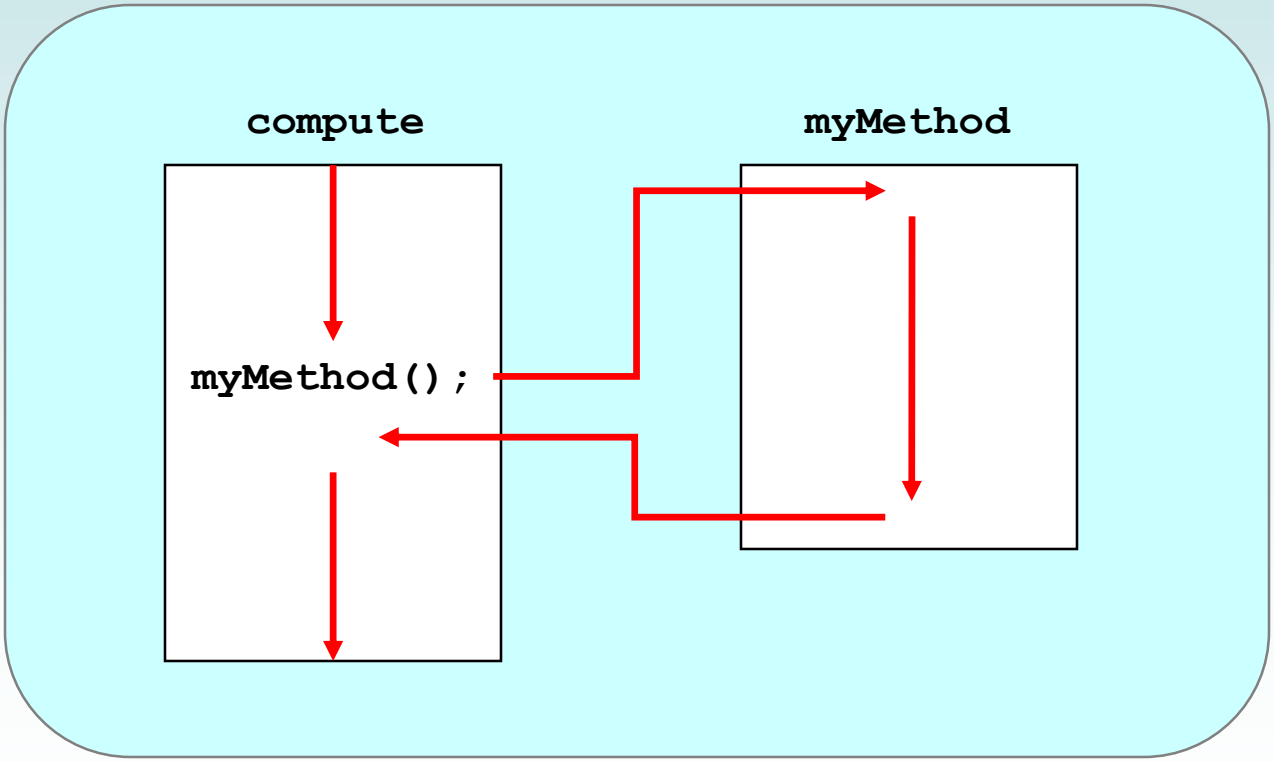
☺ **Buttons and Text Fields**

Method Declarations

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

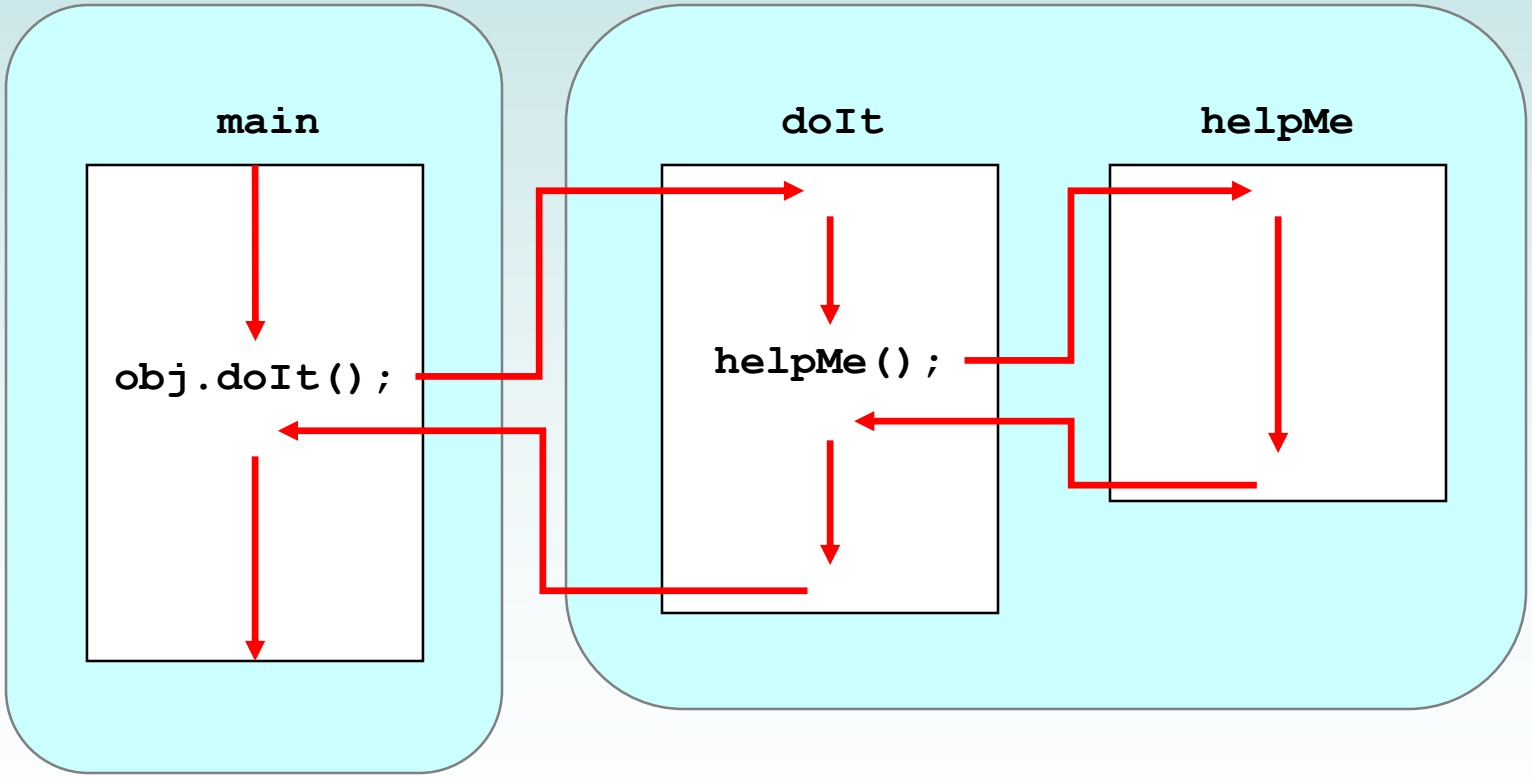
Method Control Flow

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



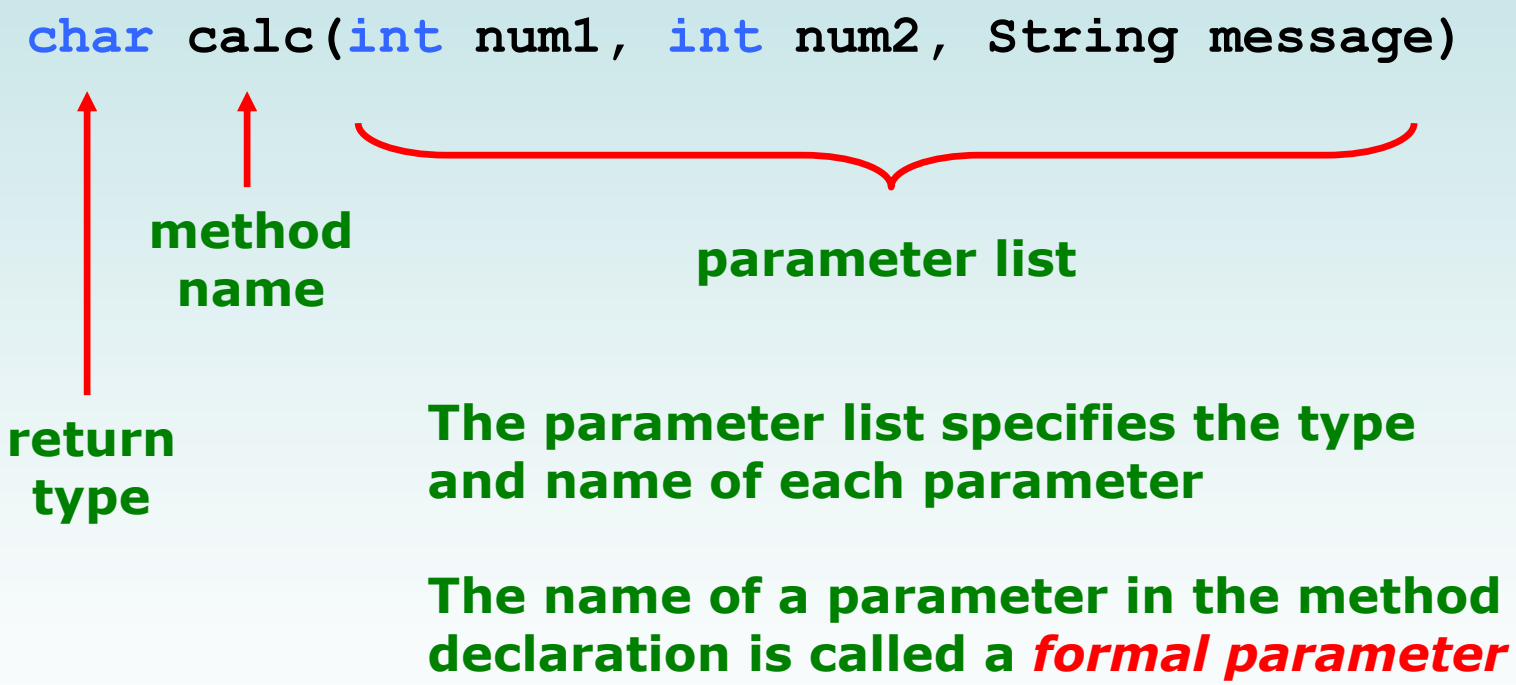
Method Control Flow

- The called method is often part of another class or object
- Use a reference to that object to call its method



Method Header

- A method declaration begins with a *method header*



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 variables**

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 is 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;
```

- `expression` must conform to the return type

Parameters

- Call by value: 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");
```

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

    return result;
}
```

Local Data

- Local variables are declared inside a method
  - but the variables don't actually exist until the method is called
- When a method is invoked, *automatic local variables* are created for the formal parameters and local variables
  - like getting a phone call and taking out a clean sheet of paper to write things down on and to calculate with
  - the parameters are the data you are given
  - the local variables are used for calculation

Local Data

- When the method finishes, all local variables are destroyed (including the formal parameters)
  - finished with the phone call:
    - tell the caller the result of calculation
    - crumple up the paper and toss it in the waste
- Instance variables hold the state of the object and exist as long as the object exists

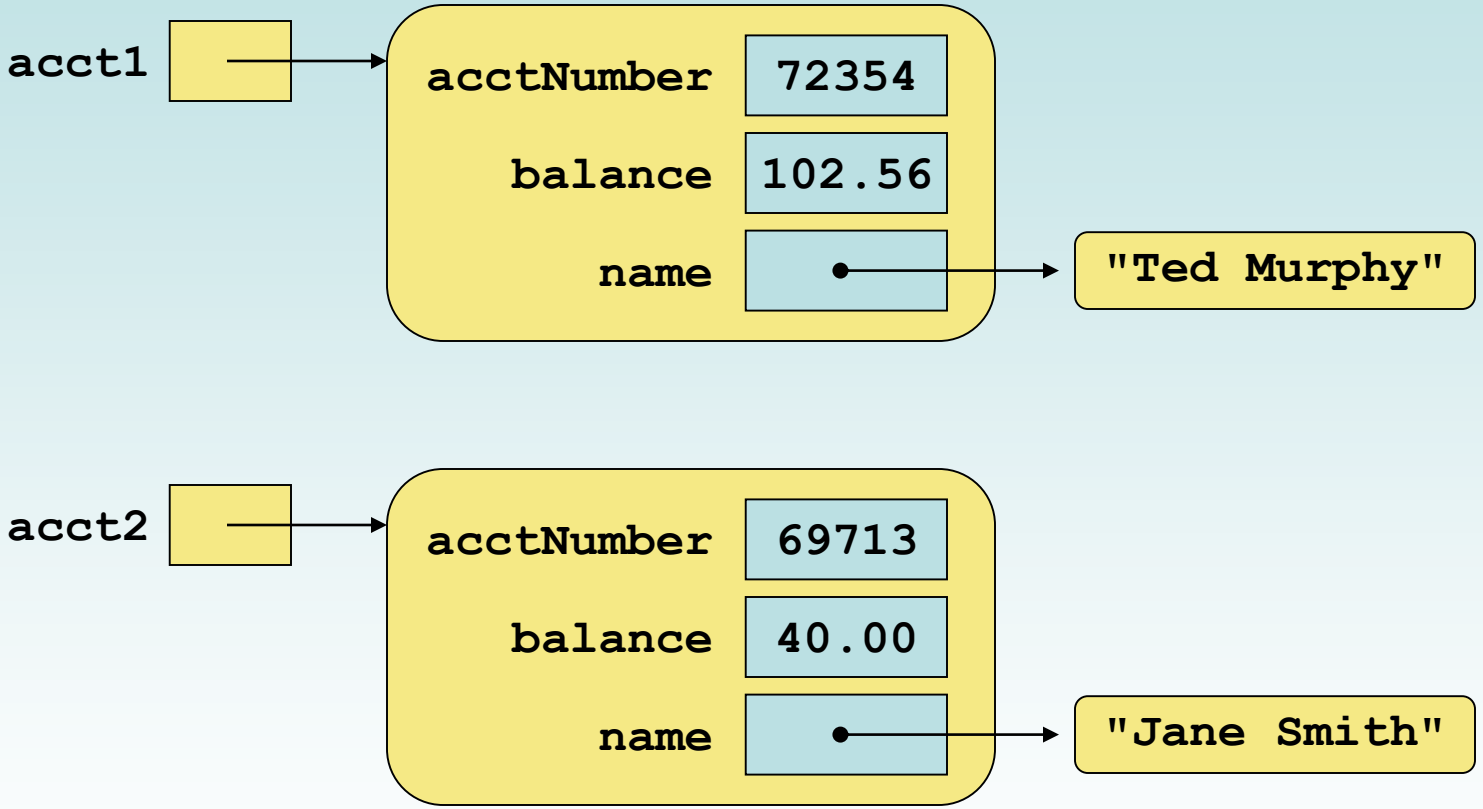
Bank Account Example

- Represent a bank account
  by a class named `Account`
- It's state includes the account number, the current balance, and the name of the owner
- An account's services include deposits, withdrawals, and adding interest

Driver Programs

- A *driver program* drives the use of other 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



```
//********************************************************************
//  Transactions.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of multiple Account objects.
//********************************************************************

public class Transactions
{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Account acct1 = new Account("Ted Murphy", 72354, 102.56);
      Account acct2 = new Account("Jane Smith", 69713, 40.00);
      Account acct3 = new Account("Edward Demsey", 93757, 759.32);

      acct1.deposit(25.85);  // ignore returned value of deposit()

      double smithBalance = acct2.deposit(500.00);
      System.out.println("Smith balance after deposit: " + smithBalance);
```

```
      System.out.println("Smith balance after withdrawal: " +
                         acct2.withdraw (430.75, 1.50));

      acct1.addInterest();
      acct2.addInterest();
      acct3.addInterest();

      System.out.println();
      System.out.println(acct1);  // automatically use toString()
      System.out.println(acct2);
      System.out.println(acct3);
   }
}
```

```
      System.out.println ("Smith balance after withdrawal: " +
                          acct2.withdraw (430.75, 1.50));

      acct1.addInterest();
      acct2.addInterest();
      acct3.addInterest();

      System.out.println();
      System.out.println(acct1);
      System.out.println(acct2);
      System.out.println(acct3);
   }
}
```

Output

```
Smith balance after deposit: 540.0
Smith balance after withdrawal: 107.55

72354   Ted Murphy      $132.90
69713   Jane Smith      $111.52
93757   Edward Demsey   $785.90
```

```
//********************************************************************
//  Account.java       Author: Lewis/Loftus
//
//  Represents a bank account with basic services such as deposit
//  and withdraw.
//********************************************************************

import java.text.NumberFormat;

public class Account
{
   private final double RATE = 0.035;  // interest rate of 3.5%

   private long   acctNumber;
   private double balance;
   private String name;

   //-----------------------------------------------------------------
   //  Sets up the account by defining its owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account(String owner, long account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }
```

```
//-----------------------------------------------------------------
//  Deposits the specified amount into the account. Returns the
//  new balance.
//-----------------------------------------------------------------
public double deposit(double amount)
{
   balance = balance + amount;
   return balance;
}

//-----------------------------------------------------------------
//  Withdraws the specified amount from the account and applies
//  the fee. Returns the new balance.
//-----------------------------------------------------------------
public double withdraw(double amount, double fee)
{
   balance = balance - amount - fee;
   return balance;
}
```

```
    //-----------------------------------------------------------------
    //  Adds interest to the account and returns the new balance.
    //-----------------------------------------------------------------
    public double addInterest()
    {
       balance += (balance * RATE);
       return balance;
    }

    //-----------------------------------------------------------------
    //  Returns the current balance of the account.
    //-----------------------------------------------------------------
    public double getBalance()
    {
       return balance;
    }

    //-----------------------------------------------------------------
    //  Returns a one-line description of the account as a string.
    //-----------------------------------------------------------------
    public String toString()
    {
       NumberFormat fmt = NumberFormat.getCurrencyInstance();
       return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
    }
}
```

Improvements

- There are some improvements that can be made to the `Account` class
- 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

- 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
- Each class has a *default constructor* that accepts 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:

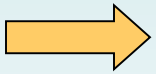
```
myAcct.deposit(50);
```

Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method



☺ Graphical Objects

☺ Graphical User Interfaces

☺ Buttons and Text Fields

☺ Graphical Objects

- Some objects contain information about how to show the object visually
  - how big to draw it, what color to use, border style…
- Most GUI components are graphical objects
- Various methods change this information
- We did this in Chapter 2 when we defined the `paint` method of an applet

☺ Smiling Face Example

- The `SmilingFace` program draws a face by defining the `paintComponent` method of a panel
- See `SmilingFace.java`
- See `SmilingFacePanel.java`
- The `main` method of the `SmilingFace` class instantiates a `SmilingFacePanel` and displays it
- The `SmilingFacePanel` class is derived from the `JPanel` class using inheritance

```
//********************************************************************
//  SmilingFace.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a separate panel class.
//********************************************************************

import javax.swing.JFrame;

public class SmilingFace
{
   //-----------------------------------------------------------------
   //  Creates the main frame of the program.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Smiling Face");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      SmilingFacePanel panel = new SmilingFacePanel();

      frame.getContentPane().add(panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```


```
//************
//  SmilingFac
//
//  Demonstrat
//************

import javax.s

public class S
{
   //--------
   //  Creates
   //--------
   public stat
   {
      JFrame frame = new JFrame("Smiling Face");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      SmilingFacePanel panel = new SmilingFacePanel();

      frame.getContentPane().add(panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  SmilingFacePanel.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a separate panel class.
//********************************************************************

import javax.swing.JPanel;
import java.awt.*;

public class SmilingFacePanel extends JPanel
{
   private final int BASEX = 120, BASEY = 60; // base point for head

   //-----------------------------------------------------------------
   //  Constructor: Sets up the main characteristics of this panel.
   //-----------------------------------------------------------------
   public SmilingFacePanel()
   {
      setBackground(Color.blue);
      setPreferredSize(new Dimension(320, 200));
      setFont(new Font("Arial", Font.BOLD, 16));
   }
```

continue

```
public void paintComponent(Graphics page)
{
      super.paintComponent(page);

      page.setColor(Color.yellow);
      page.fillOval(BASEX, BASEY, 80, 80);        // head
      page.fillOval(BASEX-5, BASEY+20, 90, 40);   // ears

      page.setColor(Color.black);
      page.drawOval(BASEX+20, BASEY+30, 15, 7);   // eyes
      page.drawOval(BASEX+45, BASEY+30, 15, 7);

      page.fillOval(BASEX+25, BASEY+31, 5, 5);    // pupils
      page.fillOval(BASEX+50, BASEY+31, 5, 5);

      page.drawArc(BASEX+20, BASEY+25, 15, 7, 0, 180);     // eyebrows
      page.drawArc(BASEX+45, BASEY+25, 15, 7, 0, 180);

      page.drawArc(BASEX+35, BASEY+40, 15, 10, 180, 180);  // nose
      page.drawArc(BASEX+20, BASEY+50, 40, 15, 180, 180);  // mouth

      page.setColor(Color.white);
      page.drawString("Always remember that you are unique!",
                           BASEX-105, BASEY-15);
      page.drawString("Just like everyone else.", BASEX-45,
                           BASEY+105);
   }
}
```

☺ Smiling Face Example

- Every Swing component has a **`paintComponent`** method
- The `paintComponent` method accepts a **`Graphics`** object that represents the graphics context for the panel
- We define the `paintComponent` method to draw the face with appropriate calls to the `Graphics` methods
- Note the difference between drawing on a panel and adding other GUI components to a panel

☺ Splat Example

- The `Splat` example is structured a bit differently
- It draws a set of colored circles on a panel, but each circle is represented as a separate object that maintains its own graphical information
- The `paintComponent` method of the panel "asks" each circle to draw itself
- See `Splat.java`
- See `SplatPanel.java`
- See `Circle.java`

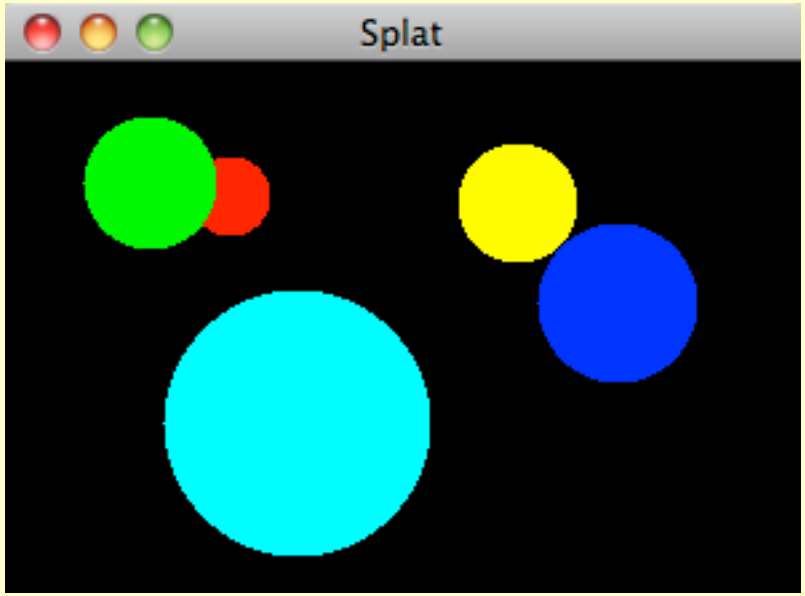
```
//********************************************************************
//  Splat.java       Author: Lewis/Loftus
//
//  Demonstrates the use of graphical objects.
//********************************************************************

import javax.swing.*;
import java.awt.*;

public class Splat
{
   //-----------------------------------------------------------------
   //  Presents a collection of circles.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Splat");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new SplatPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```




```
//*************
//  Splat.java
//
//  Demonstrate
//*************

import javax.sw
import java.awt

public class Sp
{
   //----------
   //  Presents
   //----------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Splat");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new SplatPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  SplatPanel.java       Author: Lewis/Loftus
//
//  Demonstrates the use of graphical objects.
//********************************************************************

import javax.swing.*;
import java.awt.*;

public class SplatPanel extends JPanel
{
   private Circle circle1, circle2, circle3, circle4, circle5;

   //-----------------------------------------------------------------
   //  Constructor: Creates five Circle objects.
   //-----------------------------------------------------------------
   public SplatPanel()
   {
      circle1 = new Circle( 30, Color.red,     70, 35);
      circle2 = new Circle( 50, Color.green,   30, 20);
      circle3 = new Circle(100, Color.cyan,    60, 85);
      circle4 = new Circle( 45, Color.yellow, 170, 30);
      circle5 = new Circle( 60, Color.blue,   200, 60);

      setPreferredSize(new Dimension(300, 200));
      setBackground(Color.black);
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Draws this panel by requesting that each circle draw itself.
    //-----------------------------------------------------------------
    public void paintComponent(Graphics page)
    {
        super.paintComponent(page);

        circle1.draw(page);
        circle2.draw(page);
        circle3.draw(page);
        circle4.draw(page);
        circle5.draw(page);
    }
}
```

```
//********************************************************************
//  Circle.java       Author: Lewis/Loftus
//
//  Represents a circle with a particular position, size, and color.
//********************************************************************

import java.awt.*;

public class Circle
{
   private int diameter, x, y;
   private Color color;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this circle with the specified values.
   //-----------------------------------------------------------------
   public Circle(int size, Color shade, int upperX, int upperY)
   {
      diameter = size;
      color = shade;
      x = upperX;
      y = upperY;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Draws this circle in the specified graphics context.
   //-----------------------------------------------------------------
   public void draw(Graphics page)
   {
      page.setColor(color);
      page.fillOval(x, y, diameter, diameter);
   }

   //-----------------------------------------------------------------
   //  Diameter mutator.
   //-----------------------------------------------------------------
   public void setDiameter(int size)
   {
      diameter = size;
   }

   //-----------------------------------------------------------------
   //  Color mutator.
   //-----------------------------------------------------------------
   public void setColor(Color shade)
   {
      color = shade;
   }
```

continue

continue

```
//-----------------------------------------------------------------
//  X mutator.
//-----------------------------------------------------------------
public void setX(int upperX)
{
   x = upperX;
}

//-----------------------------------------------------------------
//  Y mutator.
//-----------------------------------------------------------------
public void setY(int upperY)
{
   y = upperY;
}

//-----------------------------------------------------------------
//  Diameter accessor.
//-----------------------------------------------------------------
public int getDiameter()
{
   return diameter;
}
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Color accessor.
    //-----------------------------------------------------------------
    public Color getColor()
    {
       return color;
    }

    //-----------------------------------------------------------------
    //  X accessor.
    //-----------------------------------------------------------------
    public int getX()
    {
       return x;
    }

    //-----------------------------------------------------------------
    //  Y accessor.
    //-----------------------------------------------------------------
    public int getY()
    {
       return y;
    }
}
```

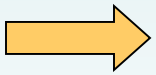
Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method

☺ Graphical Objects



☺ Graphical User Interfaces

☺ Buttons and Text Fields

☺ Graphical User Interfaces

- A Graphical User Interface (GUI) is created with three kinds of objects:
- *components* are objects that represent screen elements:
  - labels, buttons, text fields, menus, etc.
- *containers* hold and organize other components:
  - frames, panels, applets, dialog boxes
- *listener* objects respond to events
  - button clicks, mouse moves, key strikes …

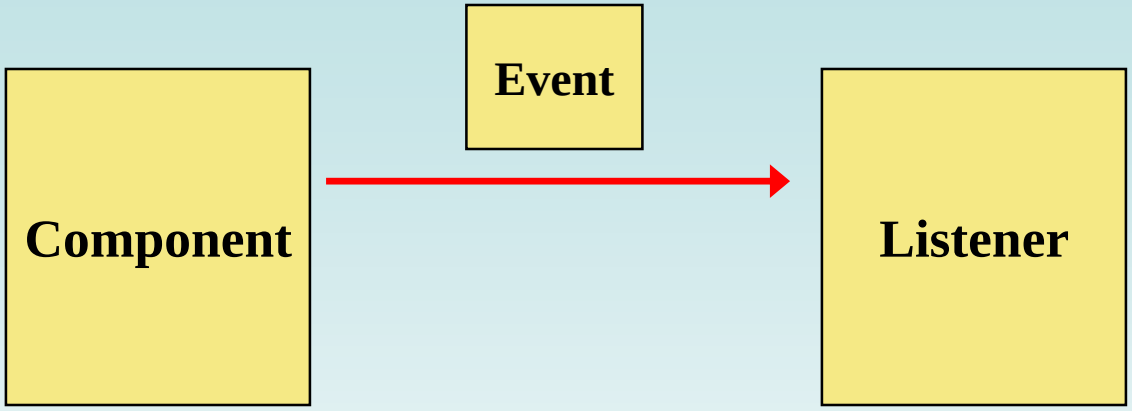
☺ Events

- An *event* is an object that represents some action involving the GUI
- We may want our program to perform some action when the following occurs:
  - the mouse is moved
  - the mouse is dragged
  - a mouse button is clicked
  - a graphical button is pressed
  - a keyboard key is pressed
  - a timer expires

☺ Events and Listeners

- The Java API contains several classes that represent events
- Components, such as a graphical button, generate (or fire) events
- A *listener* object responds to events
- We can design listener objects to take whatever actions are appropriate when an event occurs

☺ Events and Listeners



A component object generates an event

A corresponding listener object responds to the event

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

☺ GUI Development

- To create a GUI program we must:
  - set up the GUI components
  - implement listener classes for any events we care about
  - establish the relationships between listeners and the components

Outline

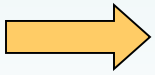
Anatomy of a Class

Encapsulation

Anatomy of a Method

☺ Graphical Objects

☺ Graphical User Interfaces



☺ Buttons and Text Fields

☺ Buttons

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

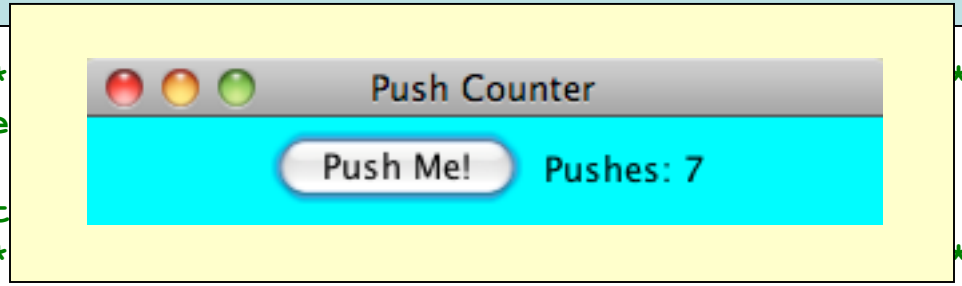
```
//********************************************************************
//  PushCounter.java       Authors: Lewis/Loftus
//
//  Demonstrates a graphical user interface and an event listener.
//********************************************************************

import javax.swing.JFrame;

public class PushCounter
{
   //-----------------------------------------------------------------
   //  Creates the main program frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Push Counter");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new PushCounterPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```




```
//*************                                              **************
//  PushCounte
//
//  Demonstrat                                               listener.
//*************                                              **************

import javax.swing.JFrame;

public class PushCounter
{
   //-----------------------------------------------------------------
   //  Creates the main program frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Push Counter");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new PushCounterPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  PushCounterPanel.java       Authors: Lewis/Loftus
//
//  Demonstrates a graphical user interface and an event listener.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PushCounterPanel extends JPanel
{
   private int count;
   private JButton push;
   private JLabel label;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the GUI.
   //-----------------------------------------------------------------
   public PushCounterPanel()
   {
      count = 0;

      push = new JButton("Push Me!");
      push.addActionListener(new ButtonListener());
```

continue

continue

```
        label = new JLabel("Pushes: " + count);

        add(push);
        add(label);

        setPreferredSize(new Dimension(300, 40));
        setBackground(Color.cyan);
    }

    //*****************************************************************
    //  Represents a listener for button push (action) events.
    //*****************************************************************
    private class ButtonListener implements ActionListener
    {
        //--------------------------------------------------------------
        //  Updates the counter and label when the button is pushed.
        //--------------------------------------------------------------
        public void actionPerformed(ActionEvent event)
        {
            count++;
            label.setText("Pushes: " + count);
        }
    }
}
```

☺ Push Counter Example

- The components of the GUI are the button, a label to display the count, a panel to organize the components, and the main frame
- The `PushCounterPanel` class represents the panel used to display the button and label
- The `PushCounterPanel` class is derived from `JPanel` using inheritance
- The constructor of `PushCounterPanel` sets up the elements of the GUI and initializes the counter to zero

☺ Push Counter Example

- The `ButtonListener` class is the listener for the action event generated by the button
- It is implemented as an *inner class*, which means it is defined within the body of another class
  - like a separate blueprint for the kitchen of a house
- This facilitates the communication between the listener and the GUI components
- Use an inner class when there is an intimate relationship between two classes and the inner class is not needed anywhere else

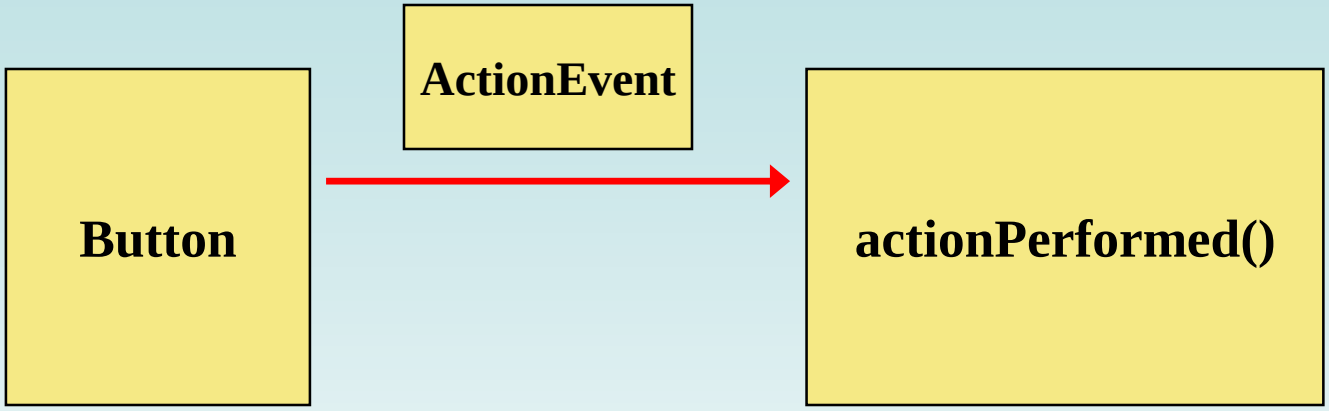
☺ Push Counter Example

- Listener classes are written by implementing a *listener interface*
- The `ButtonListener` class implements the **`ActionListener`** interface
- An interface is a list of methods that the implementing class must define
- The only method in the `ActionListener` interface is the **`actionPerformed`** method
- The Java API contains interfaces for many types of events

☺ Push Counter Example

- The `PushCounterPanel` constructor:
  - instantiates the `ButtonListener` object
  - establishes the relationship between the button and the listener by the call to **`addActionListener`**
- When the user presses the button, the button component creates an `ActionEvent` object and calls the `actionPerformed` method of the listener
- The `actionPerformed` method increments the counter and resets the text of the label

☺ Events and Listeners



A Button object generates an event

An actionListener object responds to the event

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

☺ Quick Check

Which object in the Push Counter example generated the event?

What did it do then?

☺ Quick Check

Which object in the Push Counter example generated the event?

The button component generated the event.

What did it do then?

It called the `actionPerformed` method of the listener object that had been registered with it.

☺ Text Fields

- 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 `Fahrenheit.java`
- See `FahrenheitPanel.java`

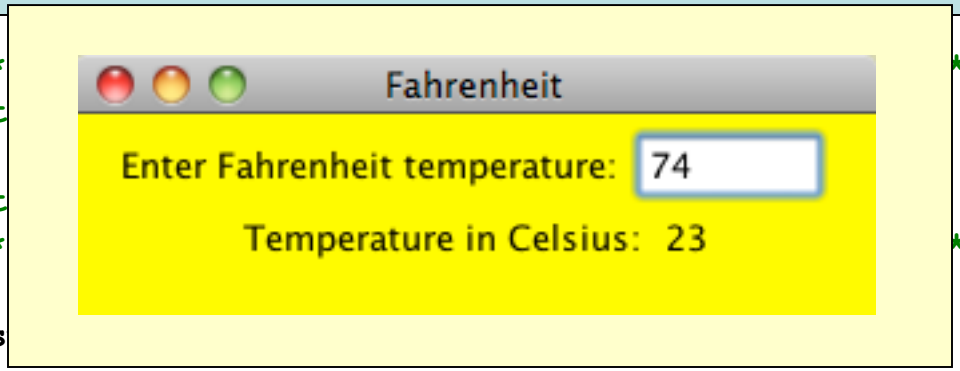
```
//********************************************************************
//  Fahrenheit.java       Author: Lewis/Loftus
//
//  Demonstrates the use of text fields.
//********************************************************************

import javax.swing.JFrame;

public class Fahrenheit
{
   //-----------------------------------------------------------------
   //  Creates and displays the temperature converter GUI.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Fahrenheit");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      FahrenheitPanel panel = new FahrenheitPanel();

      frame.getContentPane().add(panel);
      frame.pack();
      frame.setVisible(true);
   }
}
```




```
//*************                                              **************
//  Fahrenheit
//
//  Demonstrat
//*************                                              **************

import javax.s

public class Fahrenheit
{
   //-----------------------------------------------------------------
   //  Creates and displays the temperature converter GUI.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Fahrenheit");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      FahrenheitPanel panel = new FahrenheitPanel();

      frame.getContentPane().add(panel);
      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  FahrenheitPanel.java       Author: Lewis/Loftus
//
//  Demonstrates the use of text fields.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FahrenheitPanel extends JPanel
{
   private JLabel inputLabel, outputLabel, resultLabel;
   private JTextField fahrenheit;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the main GUI components.
   //-----------------------------------------------------------------
   public FahrenheitPanel()
   {
      inputLabel = new JLabel("Enter Fahrenheit temperature:");
      outputLabel = new JLabel("Temperature in Celsius: ");
      resultLabel = new JLabel("---");

      fahrenheit = new JTextField(5);
      fahrenheit.addActionListener(new TempListener());
```

```
        add(inputLabel);
        add(fahrenheit);
        add(outputLabel);
        add(resultLabel);

        setPreferredSize(new Dimension(300, 75));
        setBackground(Color.yellow);
    }

    // An inner class
    private class TempListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            int fahrenheitTemp, celsiusTemp;

            String text = fahrenheit.getText();

            fahrenheitTemp = Integer.parseInt(text);
            celsiusTemp = (fahrenheitTemp-32) * 5/9; // beware int math

            resultLabel.setText(Integer.toString(celsiusTemp));
        }
    }
}
```

☺ Fahrenheit Example

- Like the `PushCounter` example, the GUI is set up in a separate panel class
- The `TempListener` inner class defines the listener for the action event generated by the text field
- The `FahrenheitPanel` constructor instantiates the listener and adds it to the text field
- When the user types a temperature and presses enter, the text field generates the action event and calls the `actionPerformed` method of the listener

Summary

- Chapter 4 focused on:
  - class definitions
  - instance data
  - encapsulation and Java modifiers
  - method declaration and parameter passing
  - constructors
  - ☺ graphical objects
  - ☺ events and listeners
  - ☺ buttons and text fields