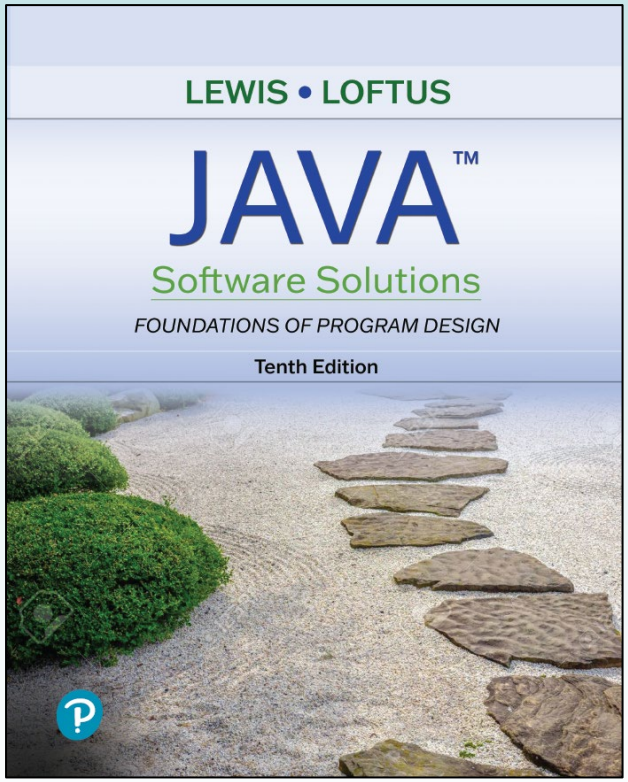
Chapter 4
Writing Classes



Java Software Solutions
Foundations of Program Design
10th Edition

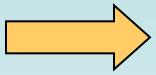
Revisions: 05/30/2024, 9/24/2025, 8/9/2026

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

Outline

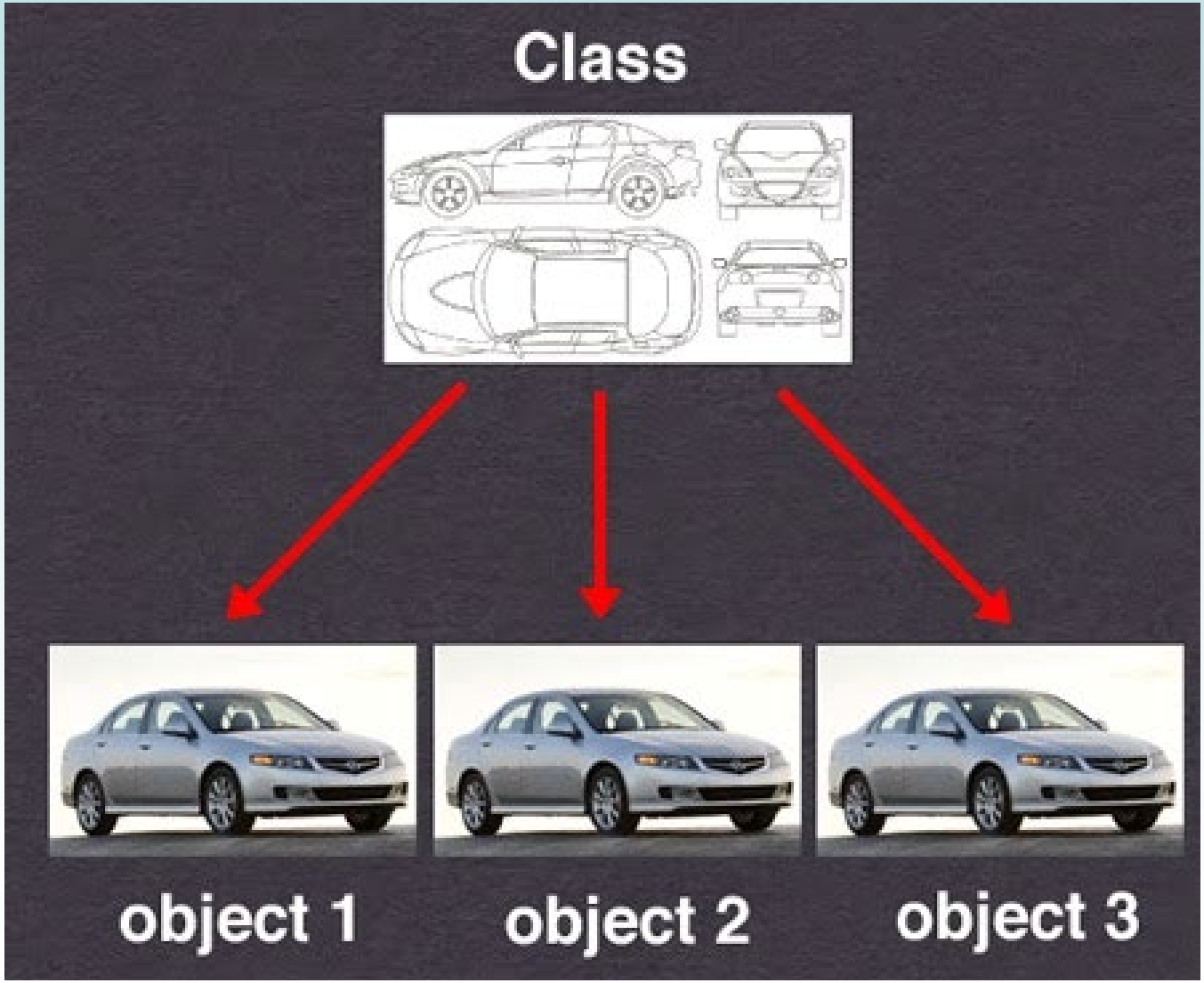


Anatomy of a Class

Encapsulation

Anatomy of a Method

§4.1 Classes and Objects



Writing Classes

- Previous examples have used classes from the Java standard class library
  - the "blueprint" already exists, all you have to do is ask for an object constructed from it
- Now we will write classes ourselves
- The class that contains the `main` method is the starting point of a program
- Object-oriented programming:
  - Create classes that represent objects
  - Use the objects to do computation

Examples of Classes

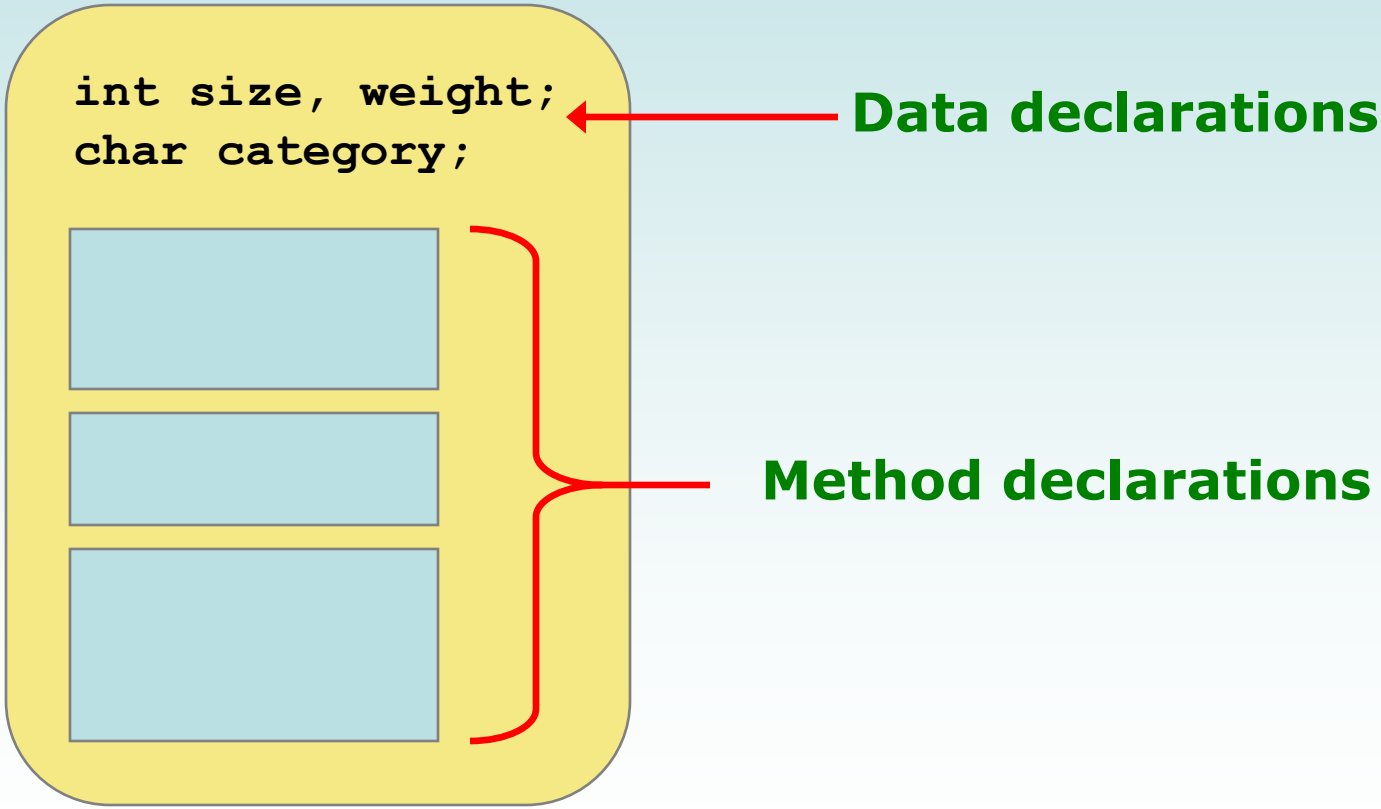
Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
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

- An object has *state* and *behavior*
- Consider a six-sided die
  - State: which face is showing
  - Behavior: die can be rolled
- We will design 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 has data declarations (state) and method declarations (behavior)



§4.2 Anatomy of a Class

- The values of its data define the state of an object
- The methods define the behaviors of the object
- Class `Die` contains an integer called `faceValue` for the current face
- A die roll sets `faceValue` to a random number between one and six

The Die Class

- `Die` objects 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 uses this instead
- Various methods of a `Die` object set and retrieve the current face value

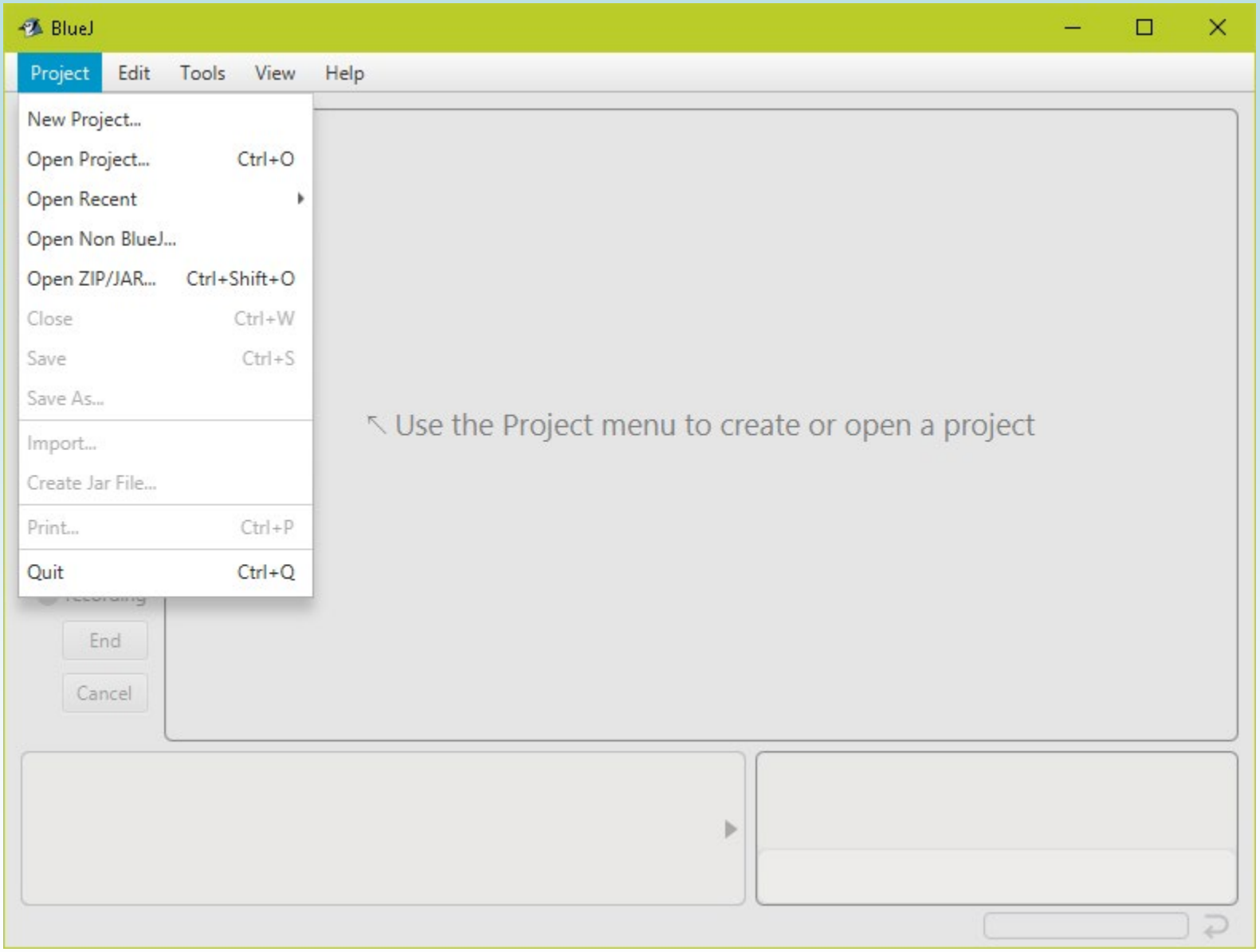
```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die with faces showing values
//  between 1 and 6.
//********************************************************************

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

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

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

```
//-----------------------------------------------------------------
//  Rolls the die and returns the result.
//  Uses random() method of Math class which
//  returns 0.0 to 0.9999...99
//-----------------------------------------------------------------
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;
}
```

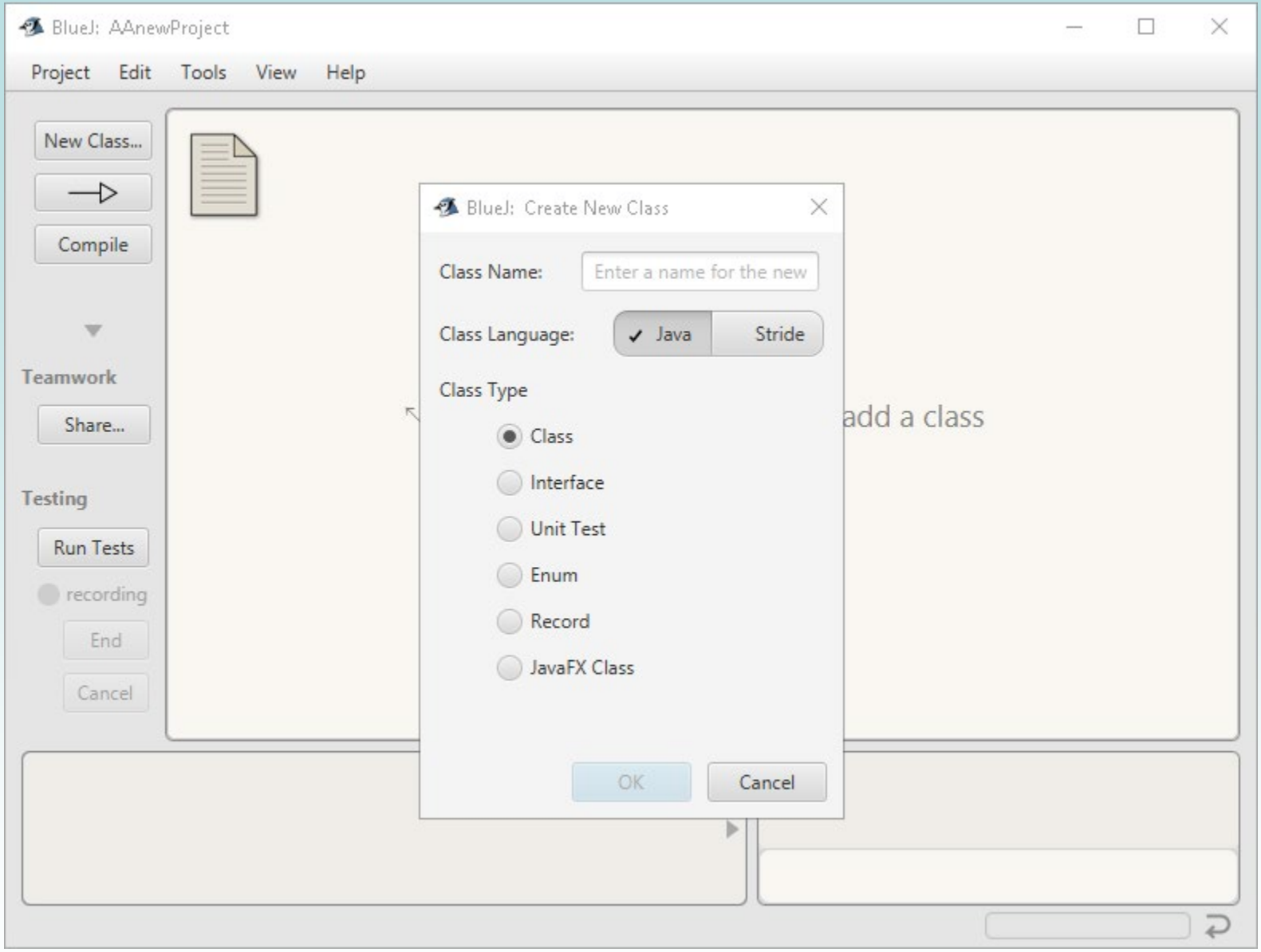
```
    //-----------------------------------------------------------------
    //  Returns a string representation of this die.
    //  Do this by concatenation with the empty string.
    //-----------------------------------------------------------------
    public String toString()
    {
       String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

       return result;
    }

} //  End of Die class
```

```
public class Die
{
   private final int MAX = 6;  // maximum face value
   private int faceValue;      // current value showing on the die

   public Die()
   {
      faceValue = 1;
   }

   public int roll()
   {
      faceValue = (int)(Math.random()*MAX) + 1;
      return faceValue;
   }

   public void setFaceValue(int value)
   {
      faceValue = value;
   }

   public int getFaceValue()
   {
      return faceValue;
   }

   public String toString()
   {
      String result =  "" + faceValue ;

      return result;
   }

} //  End of Die class
```

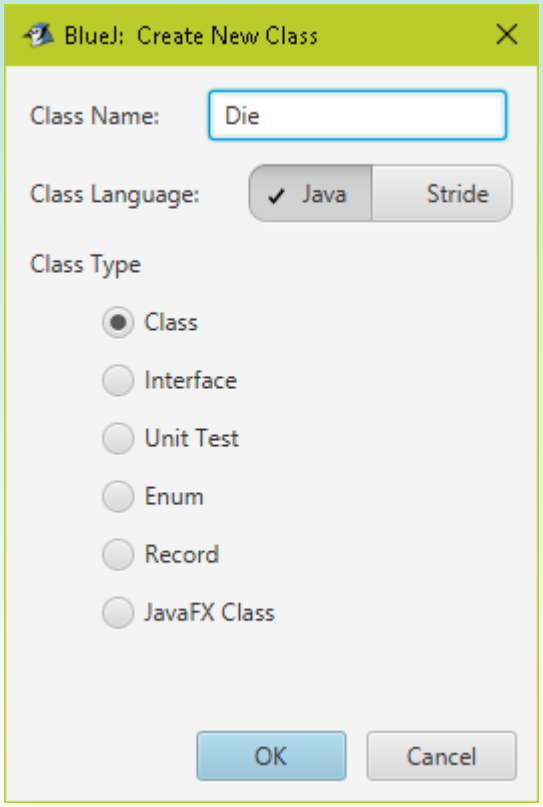
Highlight all this code and "copy" ctrl-C for later pasting.

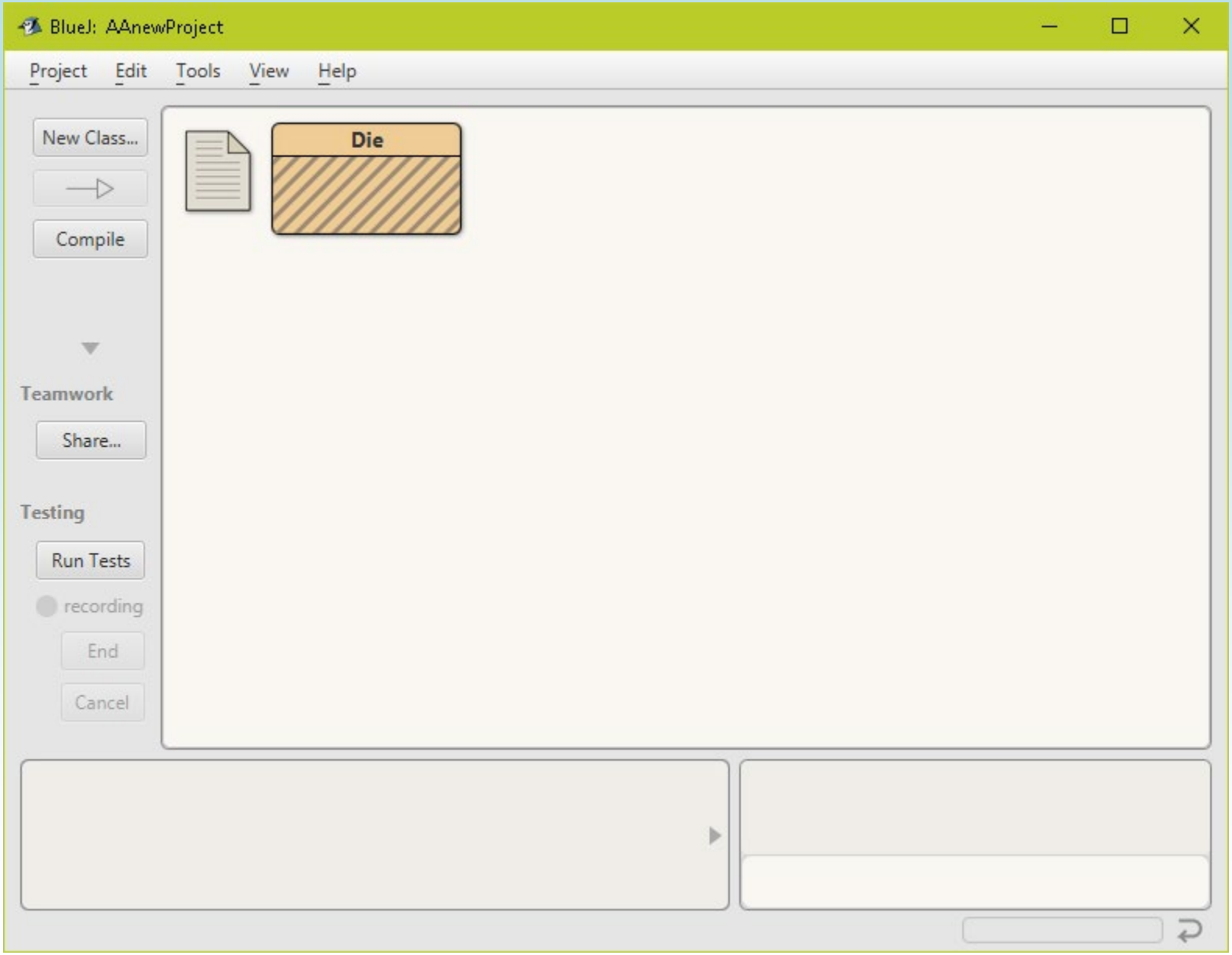


Start BlueJ and create a new Project, or re-use an existing project.

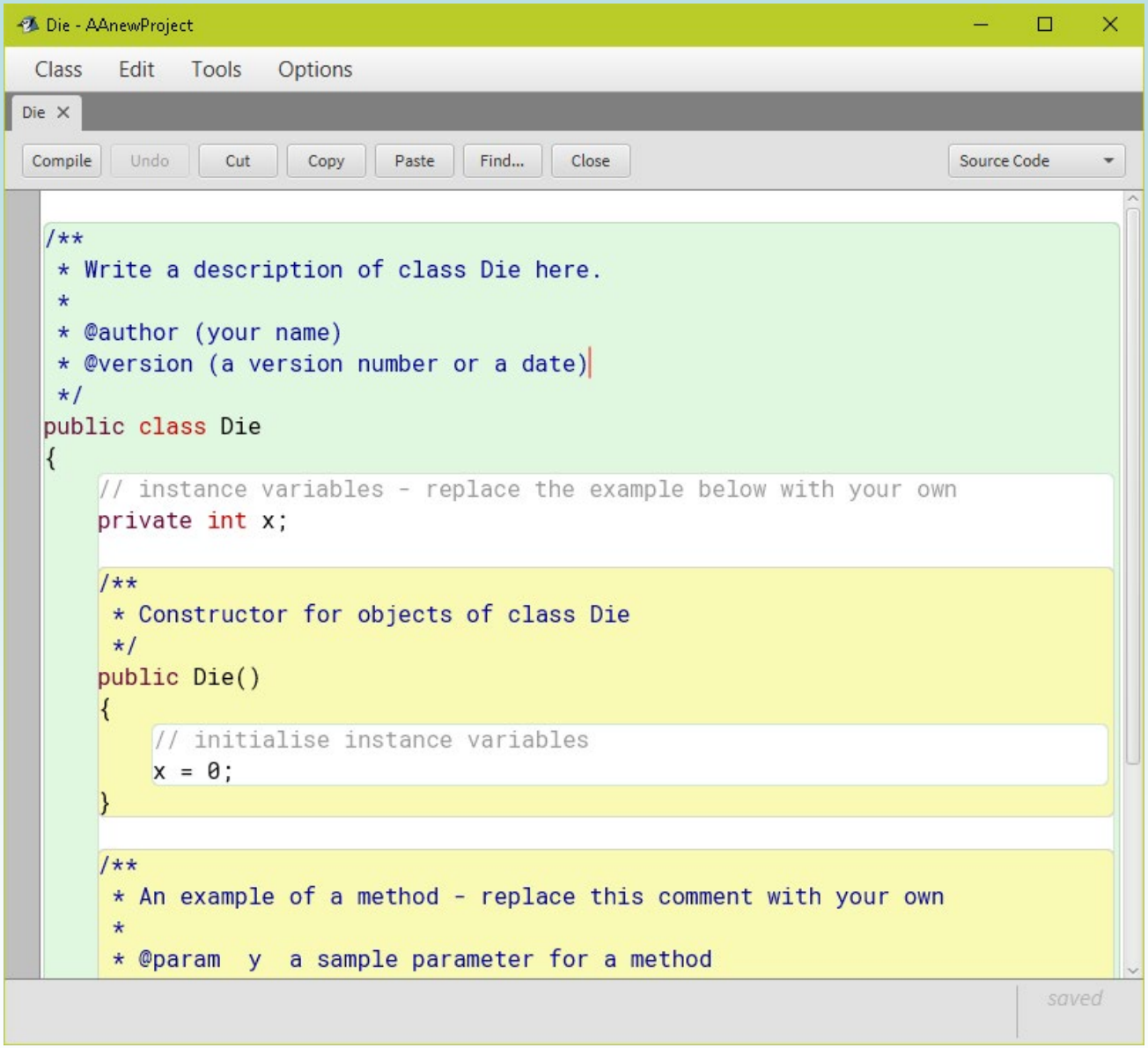


Click on "New Class." Enter Die for the class name.

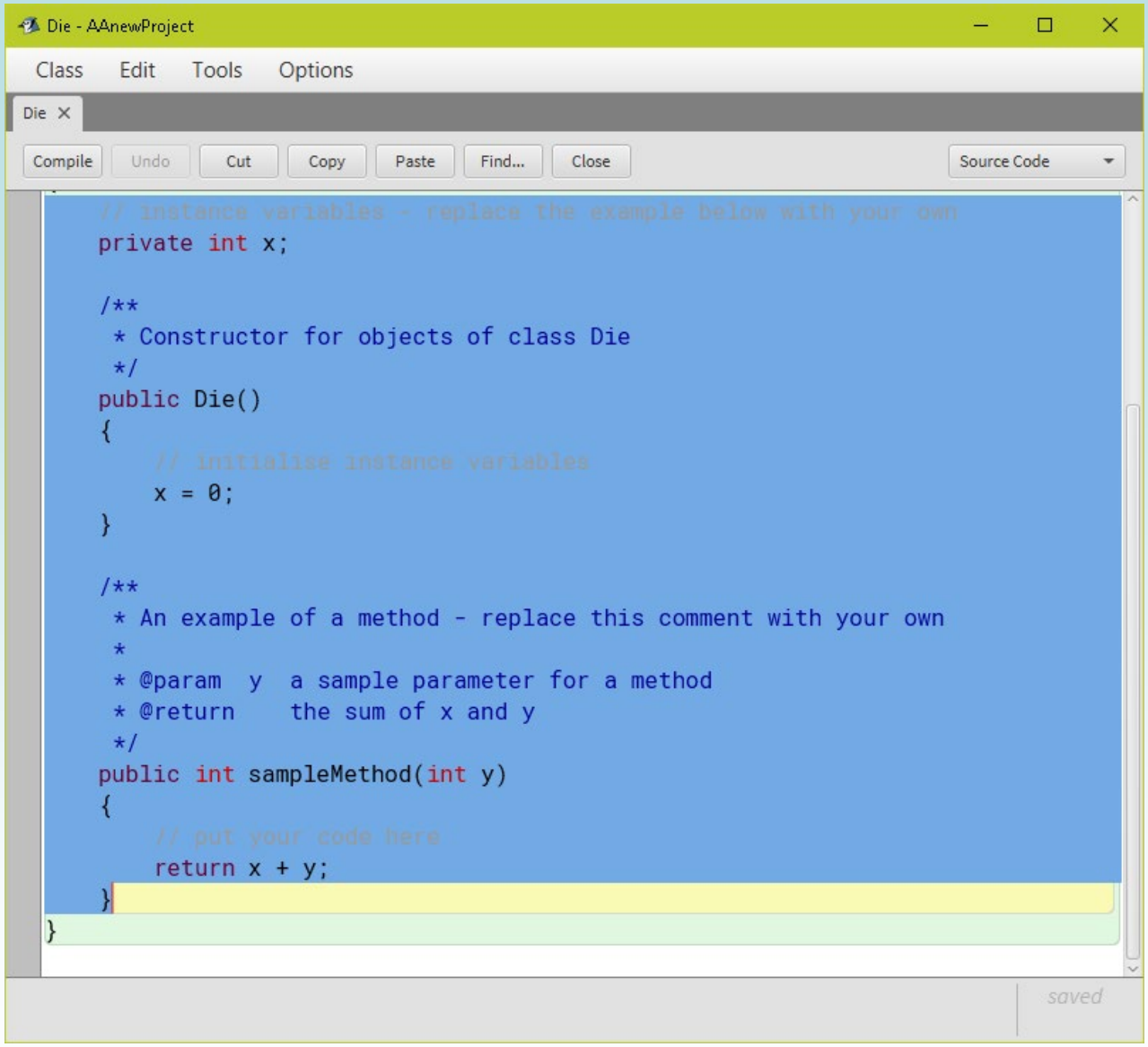

BlueJ: Create New Class
Class Name:
Die
Class Language:
Java
Stride
Class Type
Class
Interface
Unit Test
Enum
Record
JavaFX Class
OK
Cancel



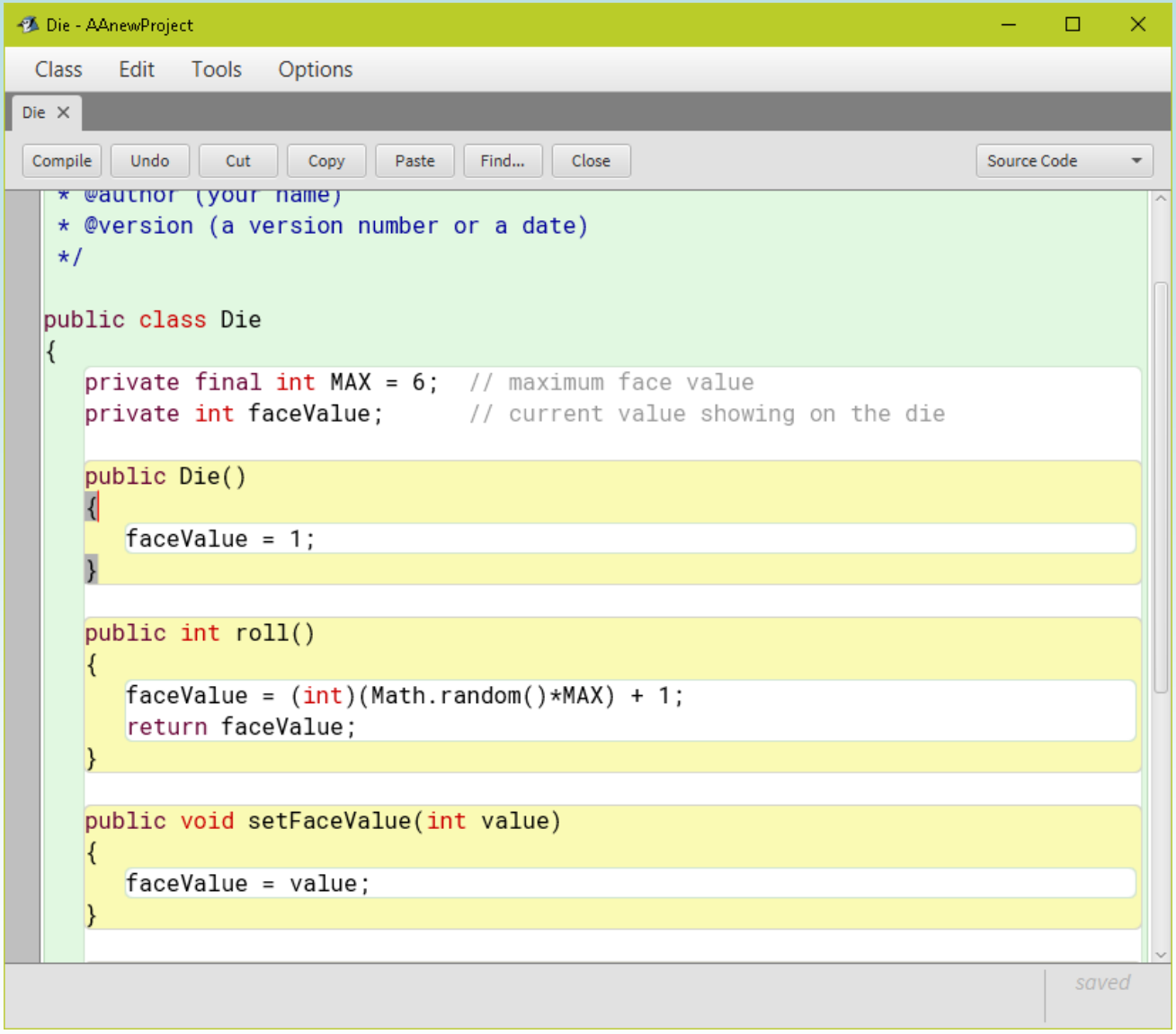
The creates the source code for Die. The box stands for it. Double left click on the box.



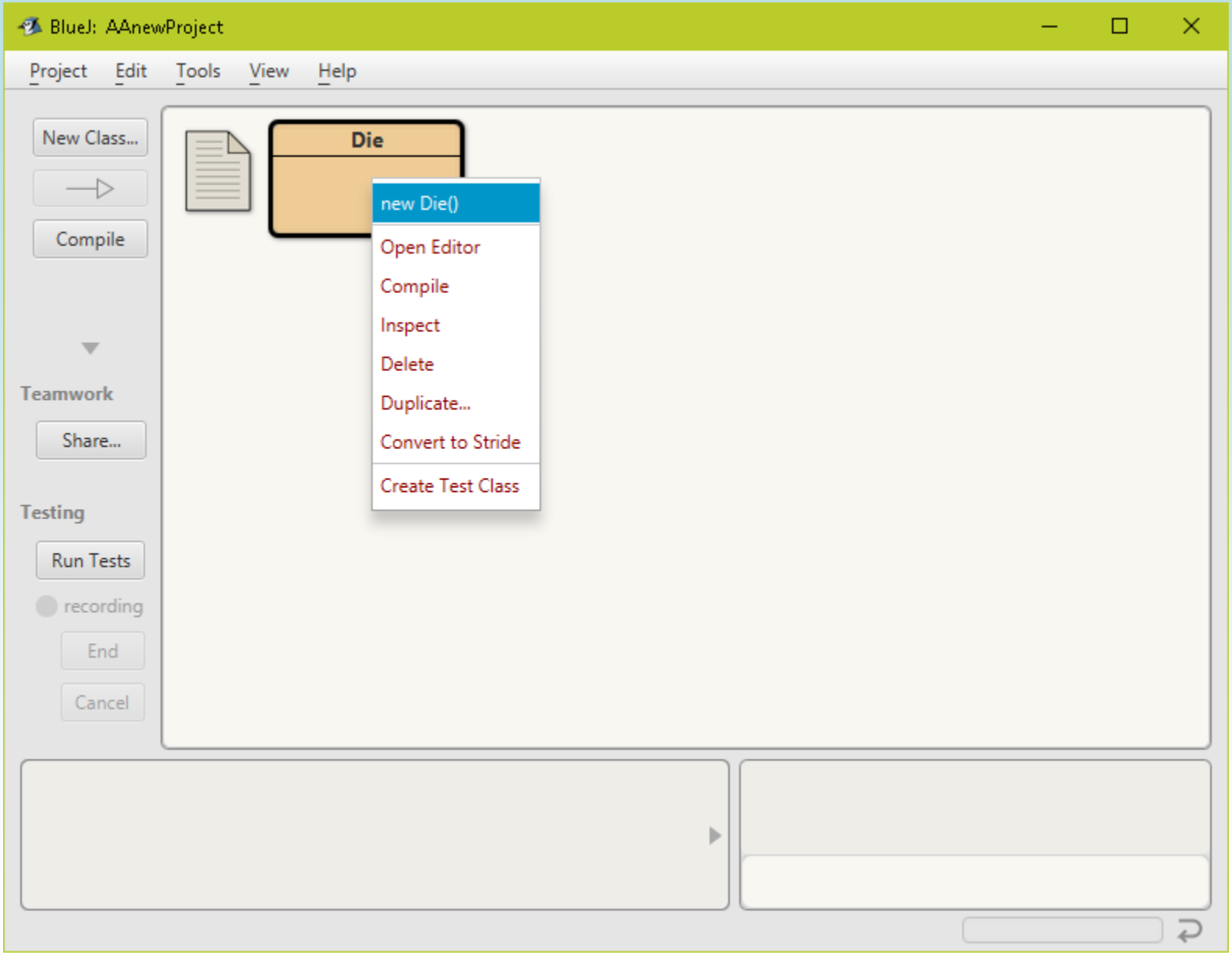
You see the automatically supplied code for Die.



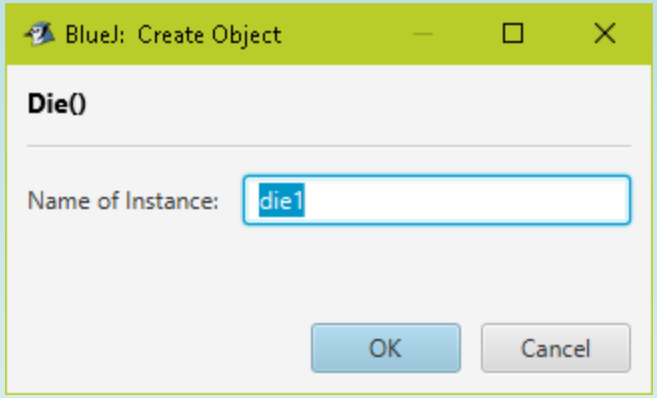
Highlight all the code. Hit delete.



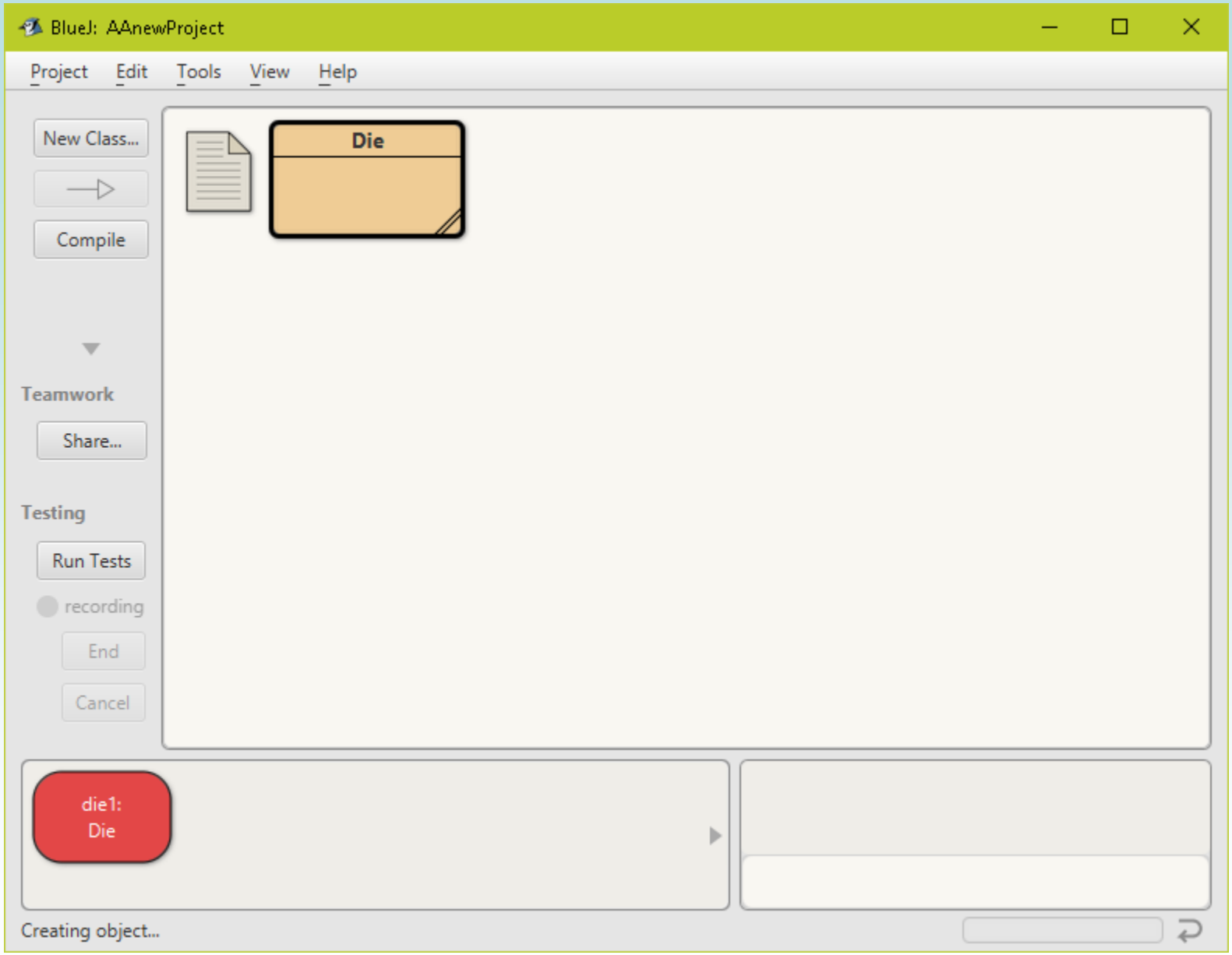
Paste the code you previously copied. Left-click "Compile."



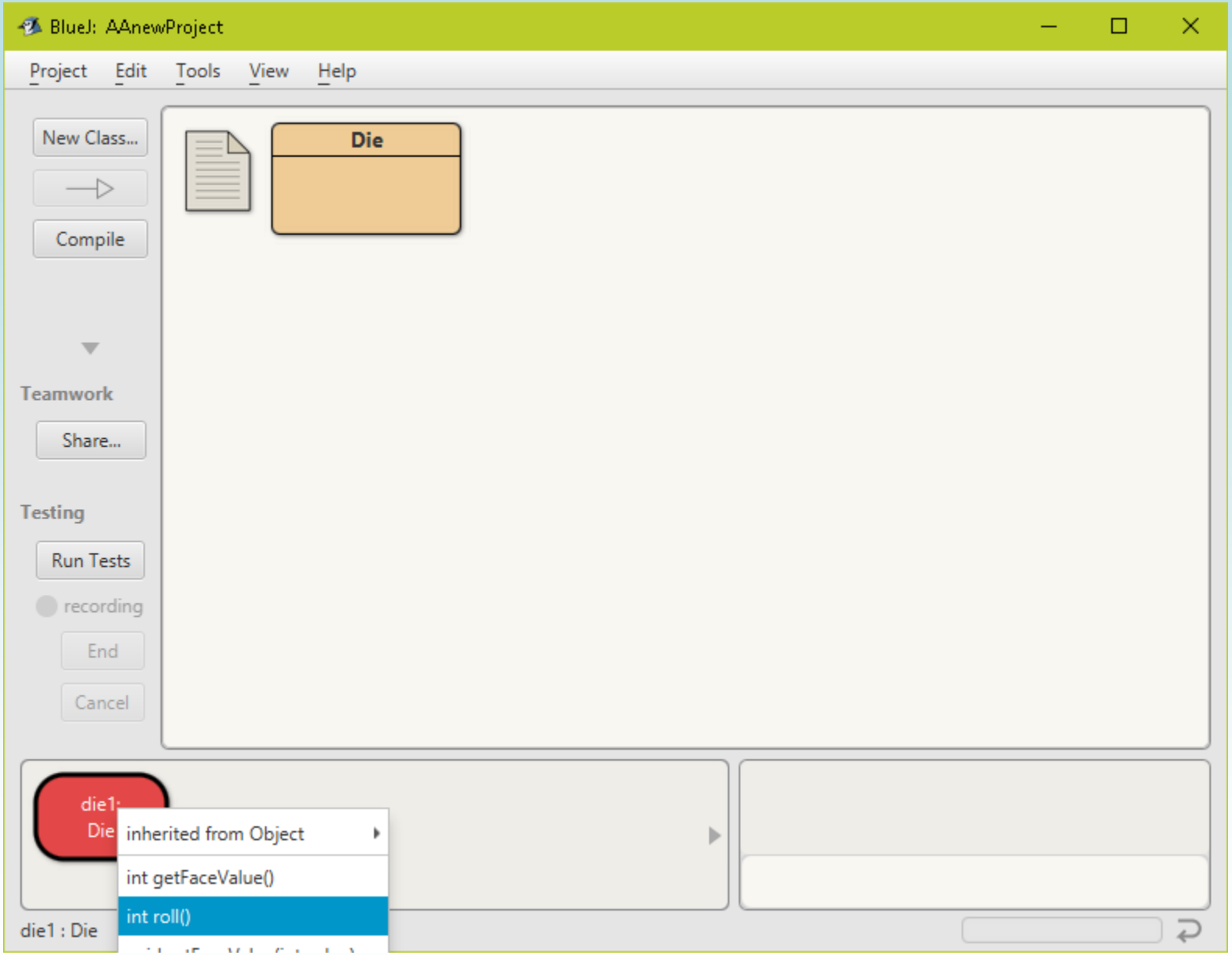
Now you have a compiled class. Right click. Pick new Die()



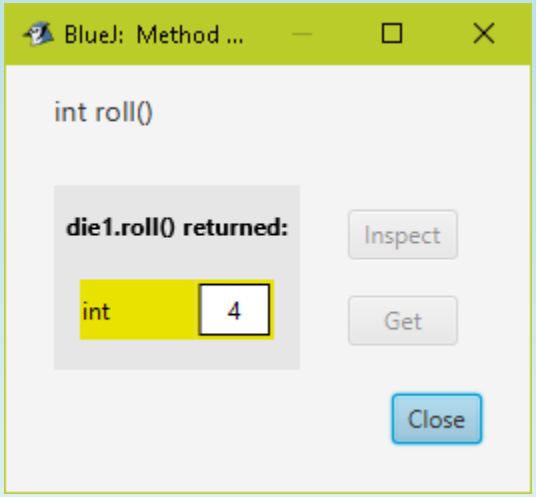
Pick a name for a reference variable for the new object (or use the automatically supplied name die1.)



The red icon represents the new Die object.



Right click on it. Select the method roll(). Left click on it.



The method runs, and returns a 4.

```
//********************************************************************
//  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);
    }
}
```

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
```

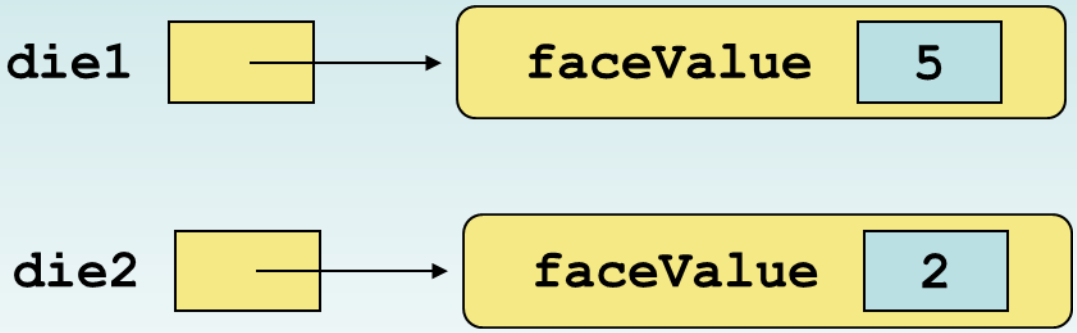
`"Die One: " + die1`

— 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.

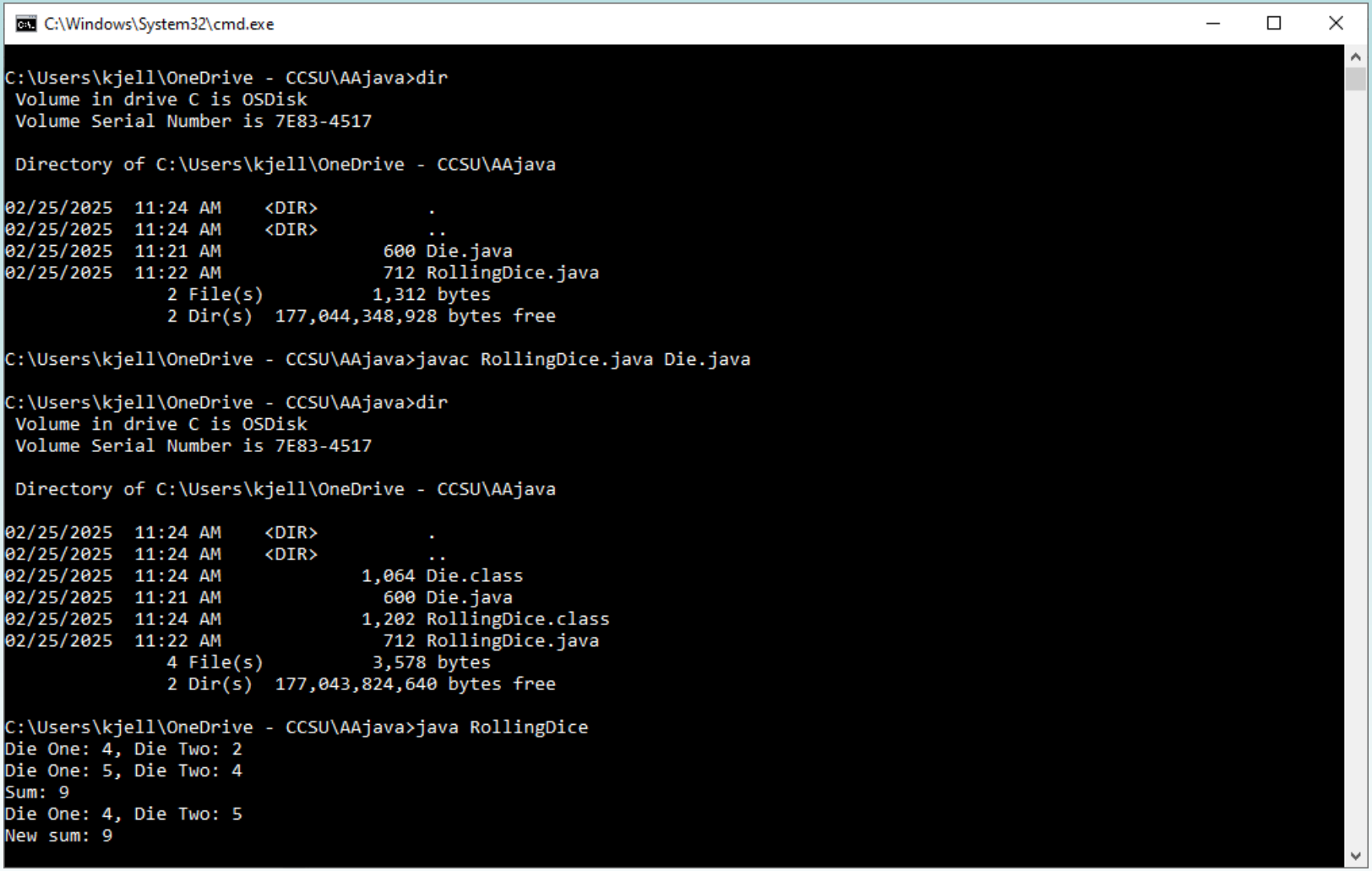
```
public class RollingDice
{
   public static void main(String[] args)
   {
      Die die1, die2;
      int sum;

      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);
   }
}
```

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



- Command line method of running the program.
- Source files Die.java and RollingDice.java Created with Notepad++
- Both files compiled together with javac
- Run the program with java using the file that contains main()

The `toString` Method

- Most classes have a `toString` method
- The `toString` method creates a `String` that represents the object and returns a reference to it
- All classes automatically have a `toString` method, but it might not do what you want
  - The Die class has a `toString` written to do what we 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 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
  - the constructor in your program just puts 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)
- An *instance variable* is declared in a class definition, but is not a part of any method
  - It is part of each object created for that class . It is part of the state of an object. Each object has its own instance variables.
  - It can be used by any method defined in that class (each object has a copy of each method)
  - `faceValue` is an instance variable
- A *local variable* is a variable declared within a method
  - A local variable can be used only by that method
  - It is “visible” only to statements within that method

- In the `Die` class, the local variable `result` is declared inside the `toString` method
  - It is local to that method and cannot be used anywhere else
  - Think of it as scratch paper the method uses
  - However, the object it points to continues to exist after control has left the method
  - The method returns a reference to that object that can be used elsewhere

```
public String toString()
{
   String result =  "" + faceValue ;

   return result;
}
```

Instance Variables

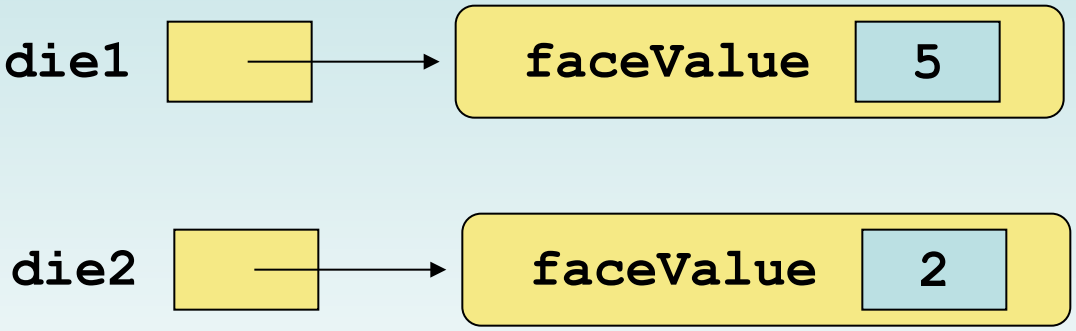
- A variable declared at the class level (such as `faceValue`) is called an *instance variable*
- Each instance (object) has its own instance variables
- A class definition describes (declares) the data, but it does not reserve memory space for it
  - that happens when an object is constructed
- When a `Die` object is created, a `faceValue` variable is created as part of it.
- The instance variables of different objects of the same class can hold different values
  - Each Die object can have its own value in its own `faceValue`

Instance Data

- The objects of a class share method definitions, but each object has its own data in instance variables
- *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 of a method and so share the actual code with other objects of the same type.

Instance Data

- The two `Die` objects from the `RollingDice` program look like this:



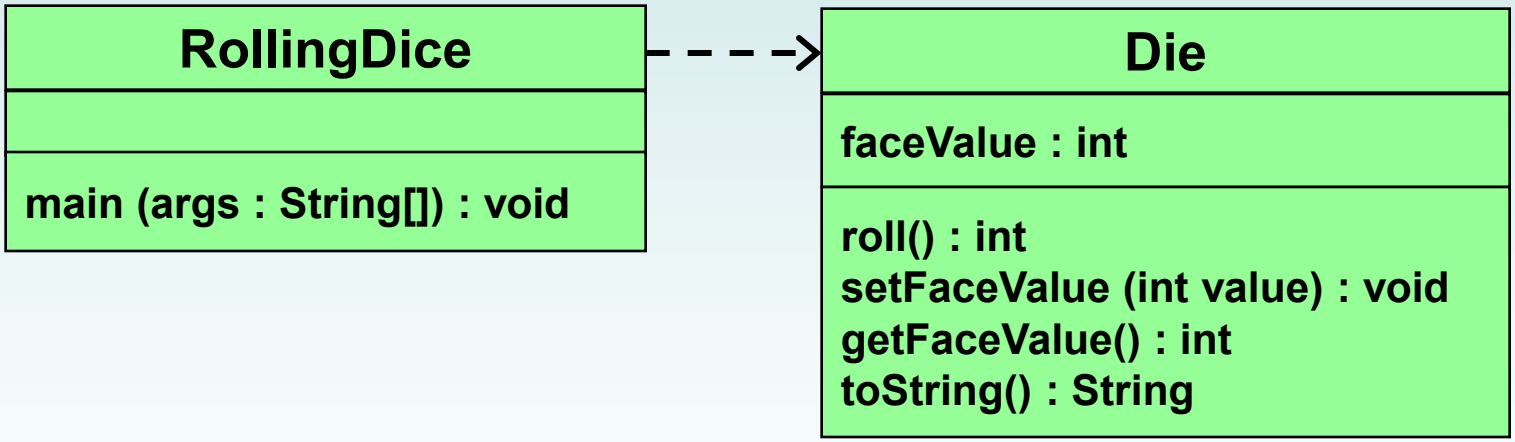
Each object maintains its own `faceValue` variable, and thus its own state

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, 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

- UML is for general software design, not only Java
- 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 describes 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.

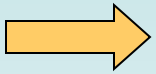
What is the scope of instance variables?

They can be referenced in any method of the class.

What are local variables?

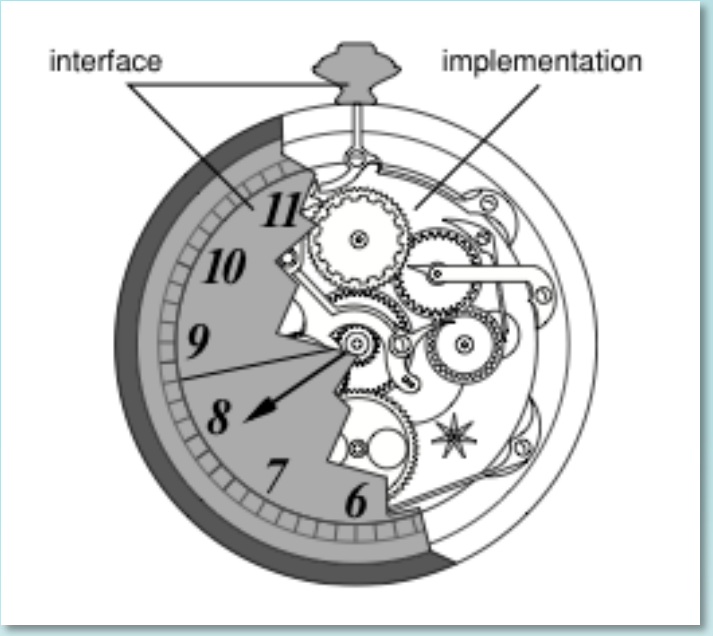
Local variables are declared within a method, and are only accessible in that method.

Outline



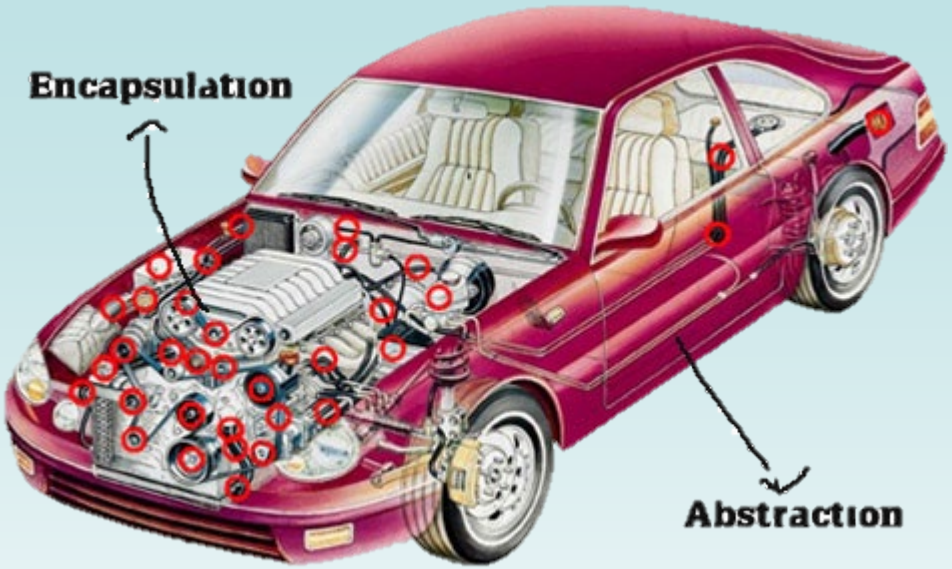
- **Anatomy of a Class**
- **Encapsulation**
- **Anatomy of a Method**
  - ☺ **Arcs**
  - ☺ **Images**
  - ☺ **Graphical User Interfaces**
  - ☺ **Text Fields**

§4.3 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 the services of another object.

We have used methods of String objects without knowing the details of how they work.

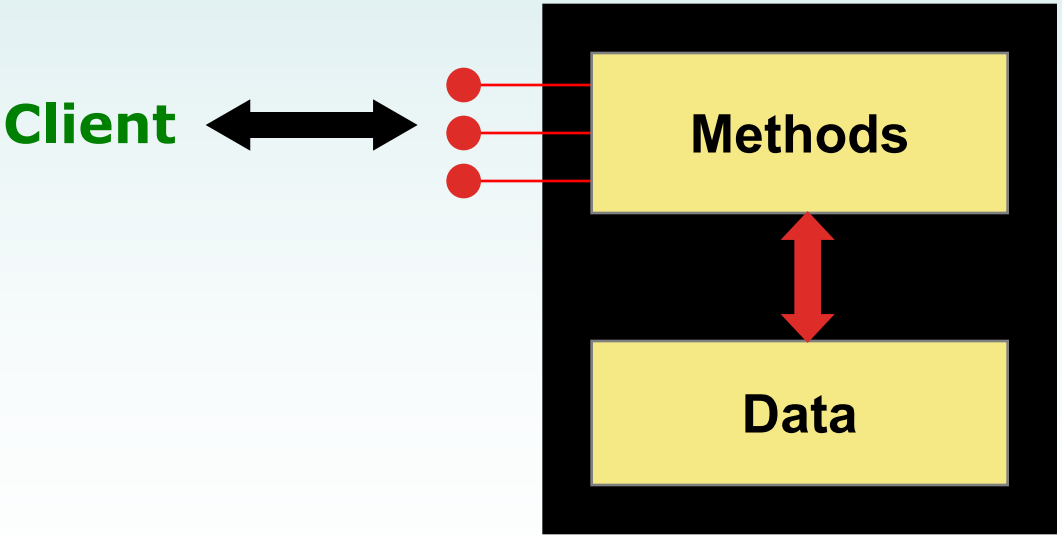
- You can use a car object without knowing the details of how it works.
  - You can twist the steering wheel, push the accelerator, hit the brakes w/o knowing the mechanisms behind those.
- Other cars should not be able to affect the inner workings of your car.

Encapsulation

- 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
- Make it difficult for a client to access an object's variables directly
  - Make variables *private*
  - Eg: *facevalue* in Die
- 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, encapsulation is done using *visibility modifiers*
- A *modifier* 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 in a later chapter

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 a client to modify the values directly
  - Like having the back of the pocket watch open
- Instance variables should be declared with private visibility
  - Eg: *facevalue* in Die
- 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 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*
  - Eg. *toss()* in Die
- A method created simply to assist a service method is called a *support method*
  - A support method is not intended to be called by a client. It should not have public visibility
  - Declare it private

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`, where `X` is the name of the value
  - Eg. *getFaceValue()* in Die
  - Eg. *setFaceValue()* in Die
- This is a convention, not syntax.
- They are sometimes called *getters* and *setters*

Mutator Restrictions

- Mutators restrict what an object's clients can do
  - 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`)
- Use if-statements for this
  - Future topic

Craziness in Cars

- Toyota cars used a computer program to control acceleration.
- Air/fuel mixture delivered to cylinders depends on many things
  - Accelerator pedal just one of them
- Program was millions of lines long, with 10,000 variables, none of them private
  - Poor modularity. Violated industry standards
- Cars suddenly accelerated for no apparent reason.
  - Driver could do nothing.
  - Dozens of deaths
  - Toyota was fined $1.3 billion
  - Similar problem with Ford cars (same software)

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.

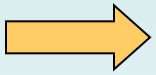
Why would it be ok to declare `MAX` as `public` in the `Die` class?

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

Outline

Anatomy of a Class

Encapsulation



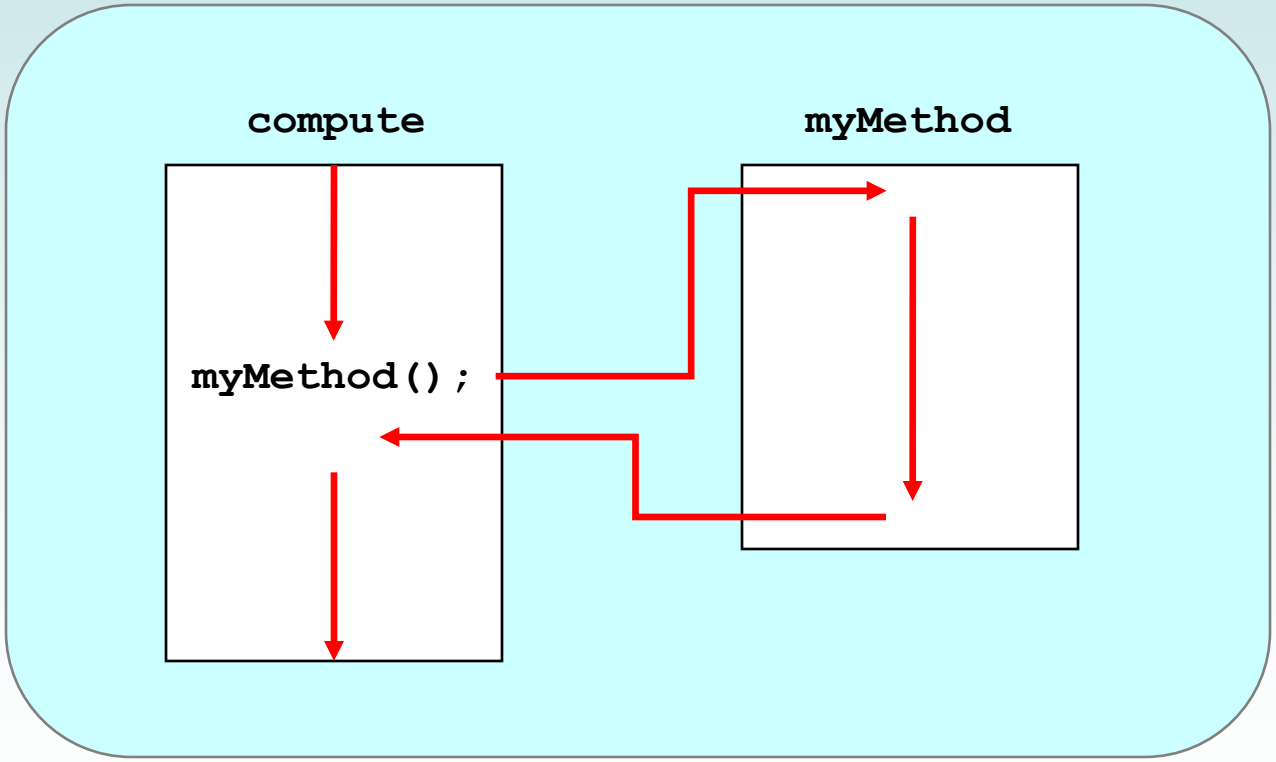
Anatomy of a Method

Method Declarations

- A *method declaration* specifies the code that makes up a method
- When a method is invoked (called), 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

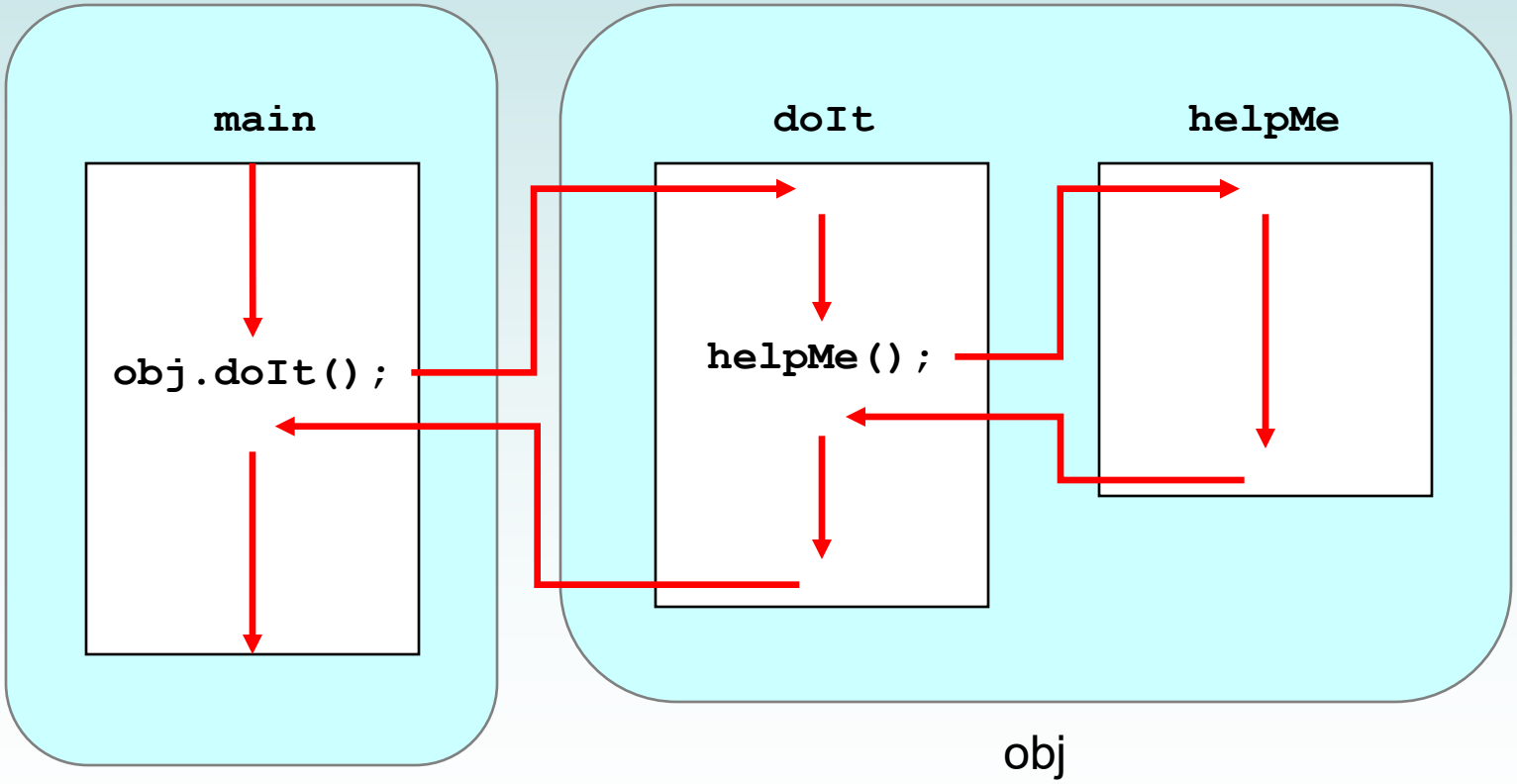
Method Control Flow

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



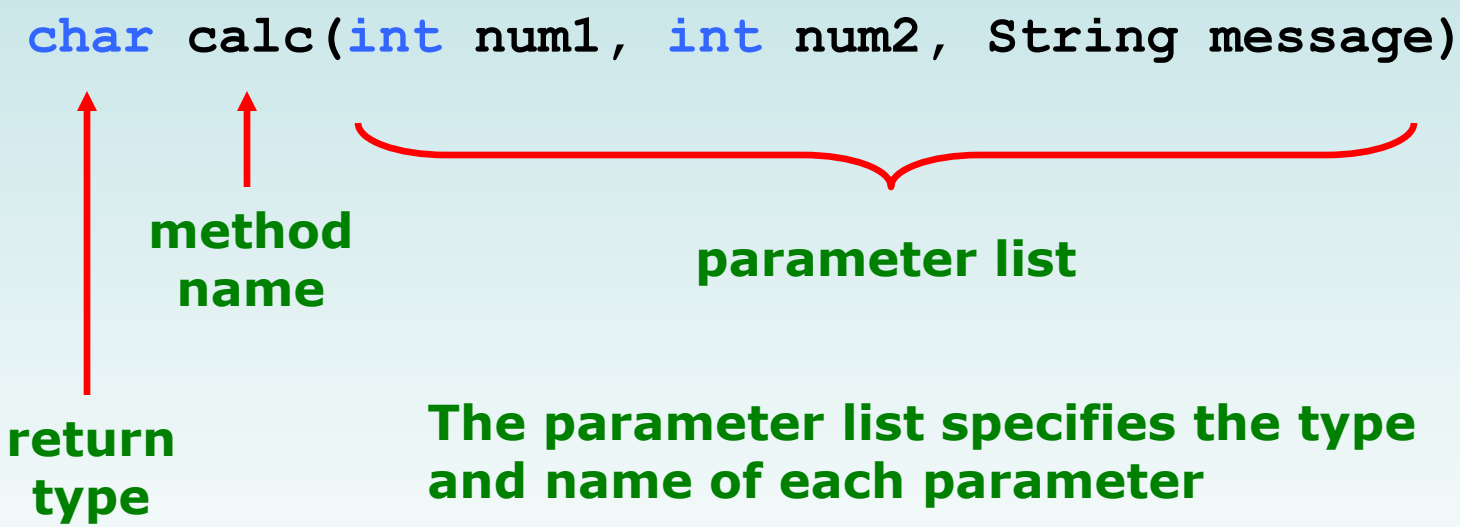
Method Control Flow

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



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

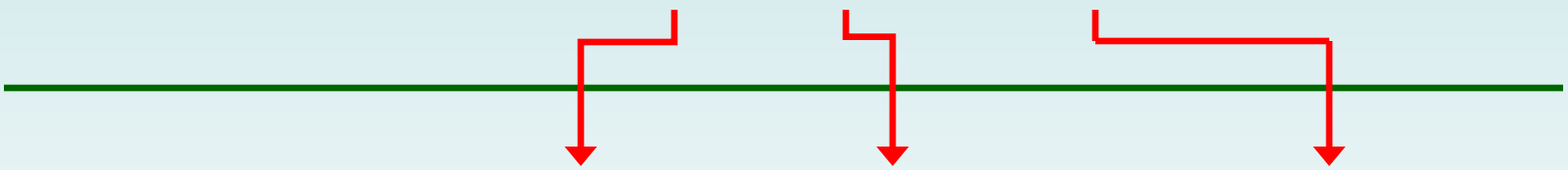
```
return expression;
```

- `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
  - The scope of a local variable is just the body of the method.
- 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 fresh 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 basket
- Instance variables are part of an object
  - They hold the state of the object and exist as long as the object exists
- Local variables are not part of an object
  - They exist for only a few moments when a method is running

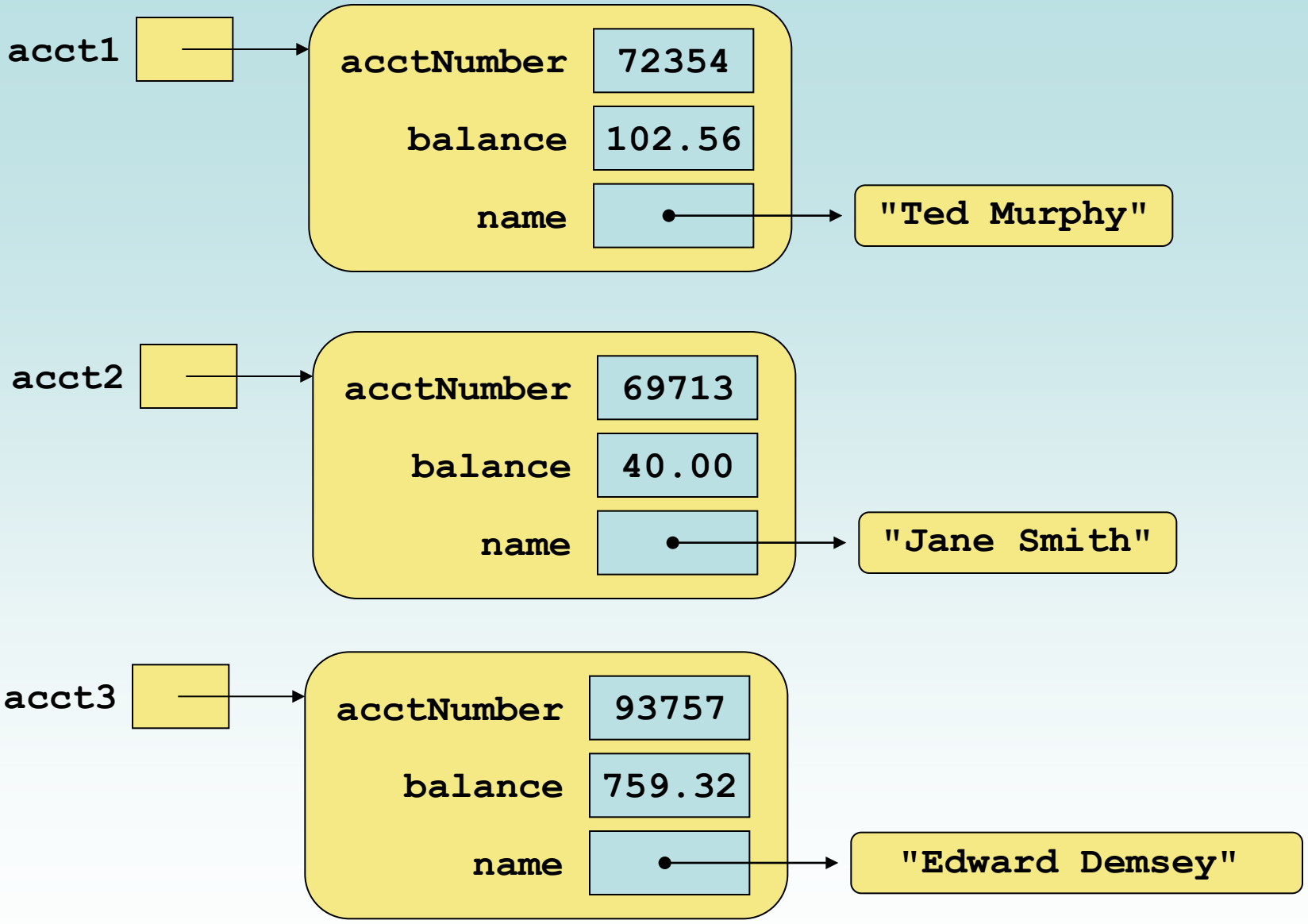
Bank Account Example

- Represent a bank account by a class named `Account`

- It's state includes
  - the account number
  - the current balance
  - the name of the owner
- It's services include
  - deposit
  - withdrawal
  - add 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`



```
//********************************************************************
//  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);
        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));
    }
}
```

```
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;

   public Account(String owner, long account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }
   public double deposit(double amount)
   {
      balance = balance + amount;
      return balance;
   }
   public double withdraw(double amount, double fee)
   {
     balance = balance - amount - fee;
     return balance;
   }
   public double addInterest()
   {
      balance += (balance * RATE);
      return balance;
   }

   public double getBalance()
   {
      return balance;
   }

   public String toString()
   {
      NumberFormat fmt = NumberFormat.getCurrencyInstance();
      return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
   }
}
```

```
public class Transactions
{
   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);
      System.out.println(acct2);
      System.out.println(acct3);
   }
}
```

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
- You do 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:

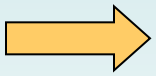
```
myAcct.deposit(50);
```

Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method



Arcs (skip)

Images (skip)

Graphical User Interfaces (skip)

Text Fields (skip)

Arcs (skip from here to end)

- 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:

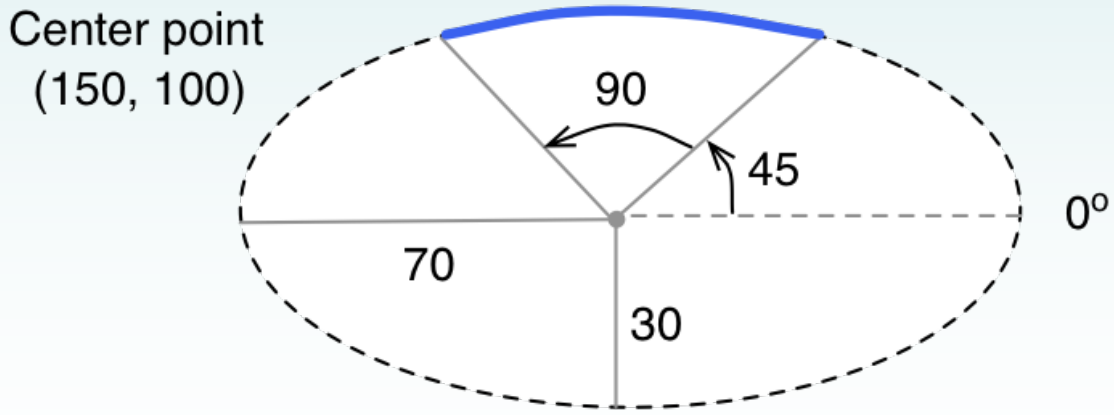
```
Arc(centerX, centerY, radiusX, radiusY,
        startAngle, arcLength)
```

- 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`

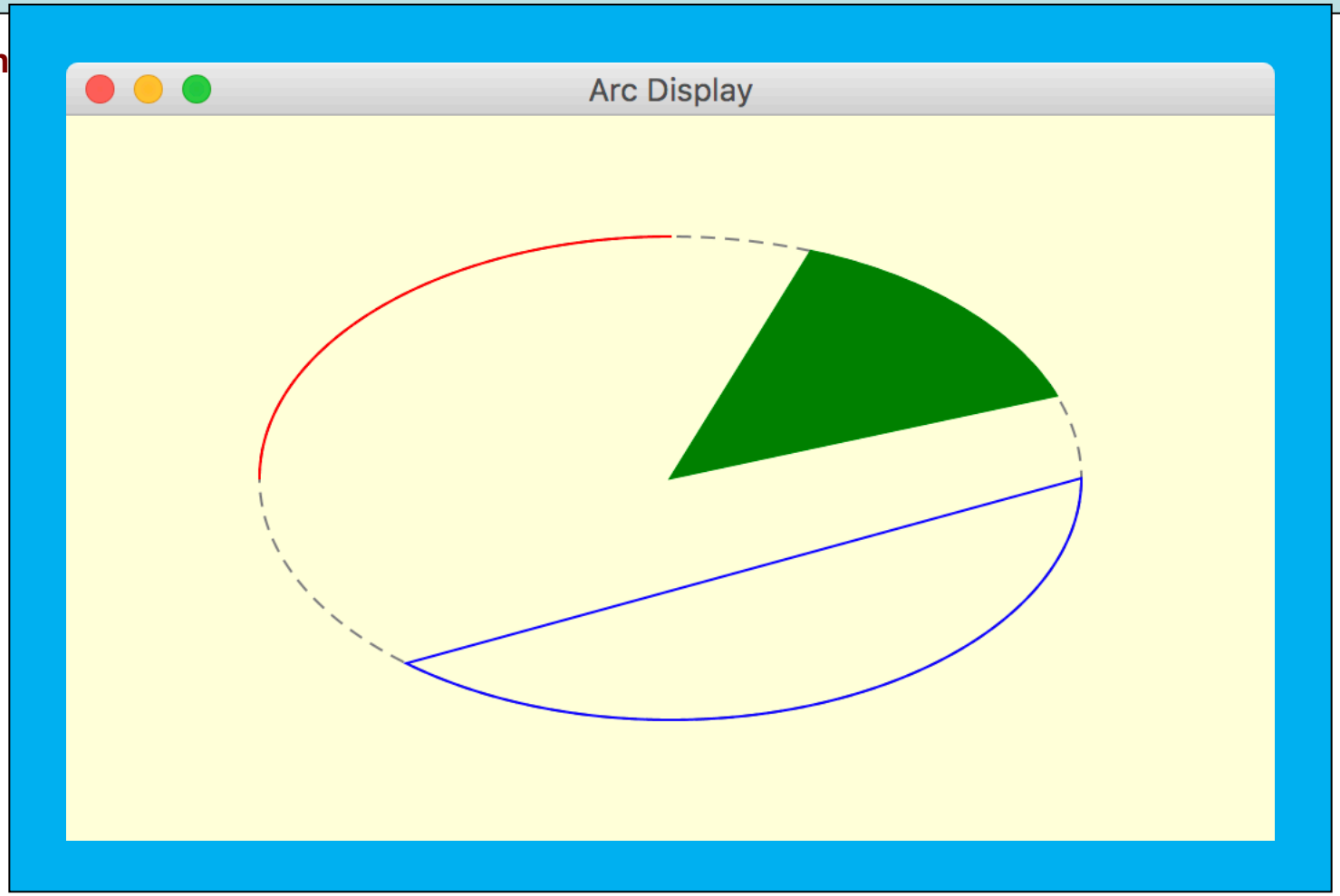
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

//*****************************************************************
//  ArcDisplay.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the JavaFX Arc class.
//*****************************************************************

public class ArcDisplay extends Application
{
    //-----------------------------------------------------------------
    //  Draws three arcs based on the same underlying ellipse.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Ellipse backgroundEllipse = new Ellipse(250, 150, 170, 100);
        backgroundEllipse.setFill(null);
        backgroundEllipse.setStroke(Color.GRAY);
        backgroundEllipse.getStrokeDashArray().addAll(5.0, 5.0);
```

continue

continue

```
        Arc arc1 = new Arc(250, 150, 170, 100, 90, 90);
        arc1.setType(ArcType.OPEN);
        arc1.setStroke(Color.RED);
        arc1.setFill(null);

        Arc arc2 = new Arc(250, 150, 170, 100, 20, 50);
        arc2.setType(ArcType.ROUND);
        arc2.setStroke(Color.GREEN);
        arc2.setFill(Color.GREEN);

        Arc arc3 = new Arc(250, 150, 170, 100, 230, 130);
        arc3.setType(ArcType.CHORD);
        arc3.setStroke(Color.BLUE);
        arc3.setFill(null);

        Group root = new Group(backgroundEllipse, arc1, arc2, arc3);
        Scene scene = new Scene(root, 500, 300, Color.LIGHTYELLOW);

        primaryStage.setTitle("Arc Display");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

contin




```
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

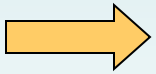
Outline

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`

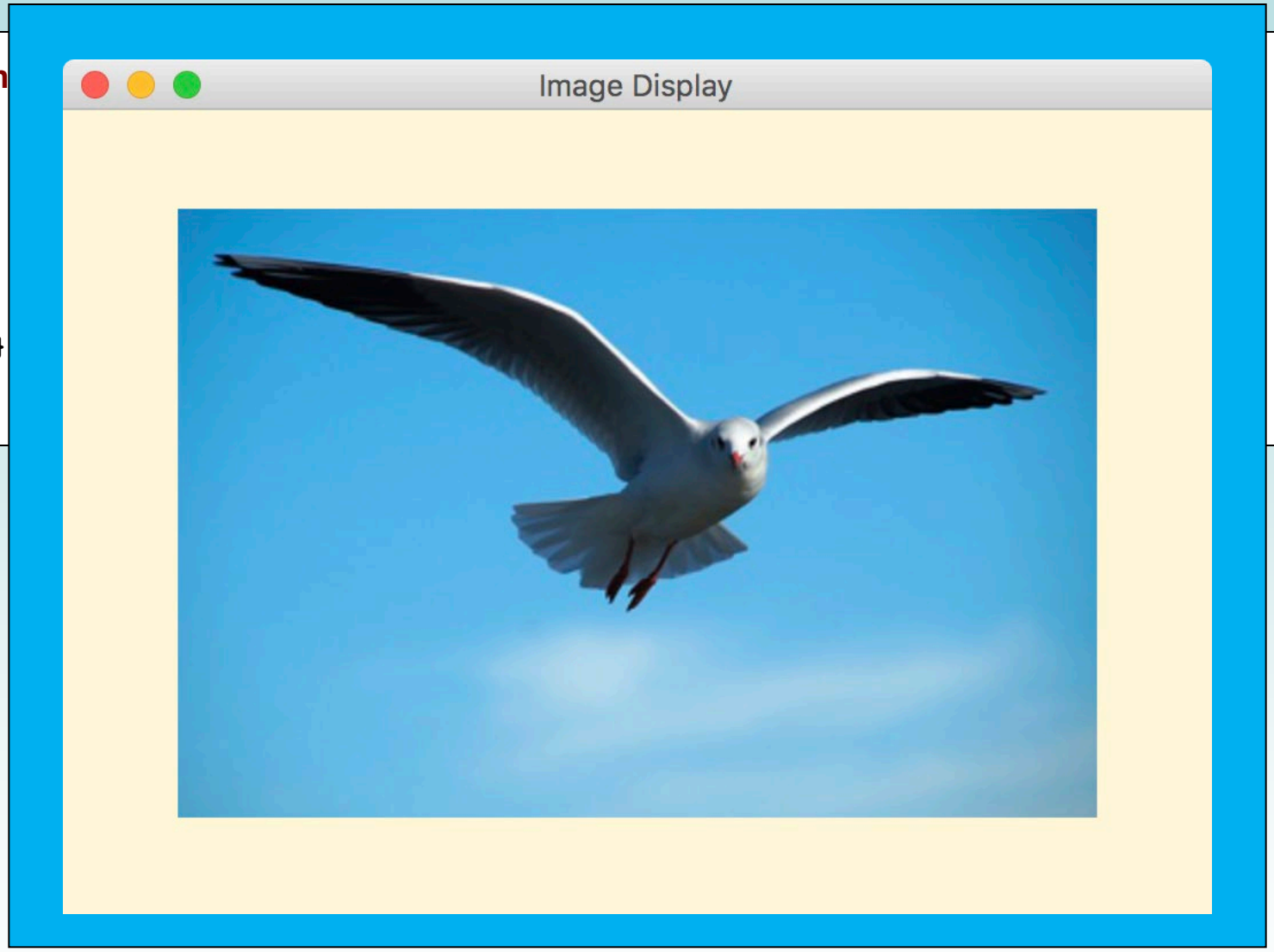
```
import javafx.application.Application;
import javafx.geometry.Rectangle2D;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

//*****************************************************************
//  ImageDisplay.java       Author: Lewis/Loftus
//
//  Demonstrates a the use of Image and ImageView objects.
//*****************************************************************

public class ImageDisplay extends Application
{
    //-----------------------------------------------------------------
    //  Displays an image centered in a window.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Image img = new Image("gull.jpg");
        ImageView imgView = new ImageView(img);

        StackPane pane = new StackPane(imgView);
        pane.setStyle("-fx-background-color: cornsilk");
```

continue

continue

```
        Scene scene = new Scene(pane, 500, 350);

        primaryStage.setTitle("Image Display");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

```
contin

        }
}
```



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

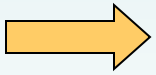
Anatomy of a Class

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.

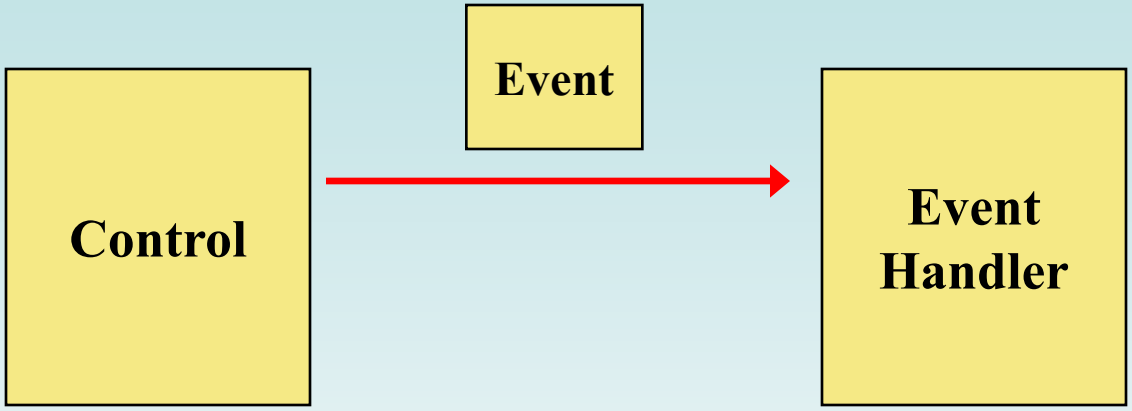
Graphical User Interfaces

- 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



A control object generates an event

A corresponding event handler is designed to respond to the event

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`

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.text.Text;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

//*****************************************************************
//  PushCounter.java       Author: Lewis/Loftus
//
//  Demonstrates JavaFX buttons and event handlers.
//*****************************************************************

public class PushCounter extends Application
{
    private int count;
    private Text countText;
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Presents a GUI containing a button and text that displays
    //  how many times the button is pushed.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        count = 0;
        countText = new Text("Pushes: 0");

        Button push = new Button("Push Me!");
        push.setOnAction(this::processButtonPress);

        FlowPane pane = new FlowPane(push, countText);
        pane.setAlignment(Pos.CENTER);
        pane.setHgap(20);
        pane.setStyle("-fx-background-color: cyan");

        Scene scene = new Scene(pane, 300, 100);

        primaryStage.setTitle("Push Counter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

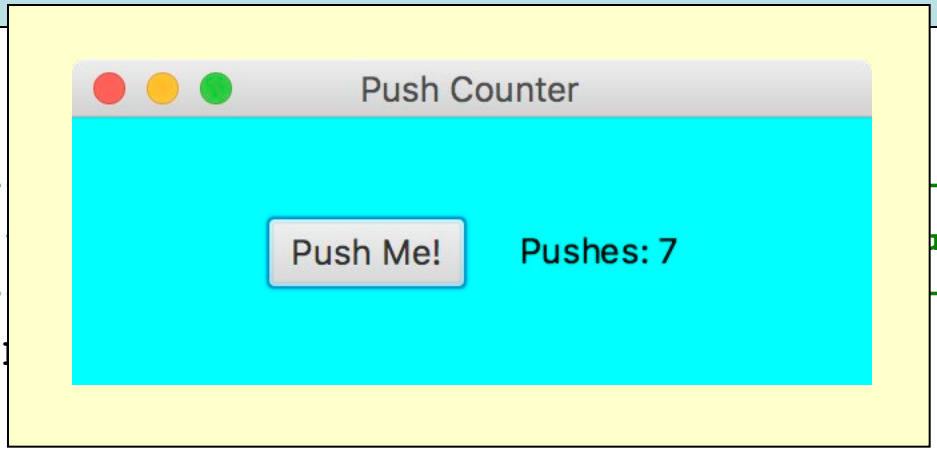
continue

continue

```
    //-----------------------------------------------------------------
    //  Updates the counter and text when the button is pushed.
    //-----------------------------------------------------------------
    public void processButtonPress(ActionEvent event)
    {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```

continue

```
    //----------                                    ----------
    //  Updates                                     ushed.
    //----------                                    ----------
    public void
    {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```



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 parmeter
- 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

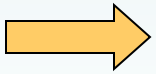
Encapsulation

Anatomy of a Method

Arcs

Images

Graphical User Interfaces

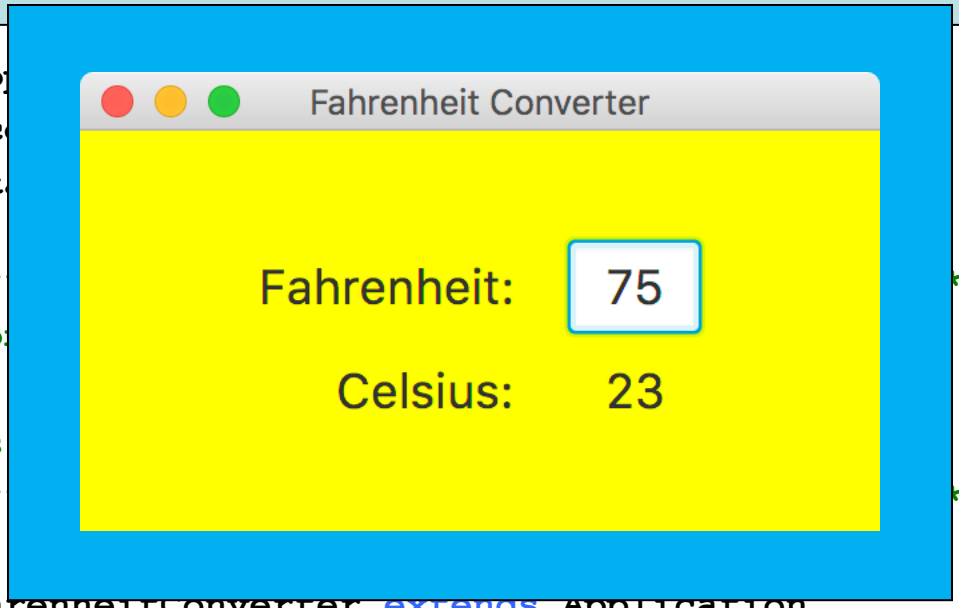


Text Fields

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`

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

//*****************************************************************
//  FahrenheitConverter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a TextField and a GridPane.
//*****************************************************************

public class FahrenheitConverter extends Application
{
    //-----------------------------------------------------------------
    //  Launches the temperature converter application.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```




```
import javafx.ap
import javafx.sc
import javafx.st

//**************                                          ***************
//  FahrenheitCo
//
//  Demonstrates
//**************                                          ***************

public class FahrenheitConverter extends Application
{
    //-----------------------------------------------------------------
    //  Launches the temperature converter application.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

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`

```
import javafx.event.ActionEvent;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;

//*******************************************************************
//  FahrenheitPane.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a TextField and a GridPane.
//*******************************************************************

public class FahrenheitPane extends GridPane
{
    private Label result;
    private TextField fahrenheit;
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Sets up a GUI containing a labeled text field for converting
    //  temperatures in Fahrenheit to Celsius.
    //-----------------------------------------------------------------
    public FahrenheitPane()
    {
        Font font = new Font(18);

        Label inputLabel = new Label("Fahrenheit:");
        inputLabel.setFont(font);
        GridPane.setHalignment(inputLabel, HPos.RIGHT);

        Label outputLabel = new Label("Celsius:");
        outputLabel.setFont(font);
        GridPane.setHalignment(outputLabel, HPos.RIGHT);

        result = new Label("---");
        result.setFont(font);
        GridPane.setHalignment(result, HPos.CENTER);
```

continue

continue

```
        fahrenheit = new TextField();
        fahrenheit.setFont(font);
        fahrenheit.setPrefWidth(50);
        fahrenheit.setAlignment(Pos.CENTER);
        fahrenheit.setOnAction(this::processReturn);

        setAlignment(Pos.CENTER);
        setHgap(20);
        setVgap(10);
        setStyle("-fx-background-color: yellow");

        add(inputLabel, 0, 0);
        add(fahrenheit, 1, 0);
        add(outputLabel, 0, 1);
        add(result, 1, 1);
    }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Computes and displays the converted temperature when the user
    //  presses the return key while in the text field.
    //-----------------------------------------------------------------
    public void processReturn(ActionEvent event)
    {
        int fahrenheitTemp = Integer.parseInt(fahrenheit.getText());
        int celsiusTemp = (fahrenheitTemp - 32) * 5 / 9;
        result.setText(celsiusTemp + "");
    }
}
```

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