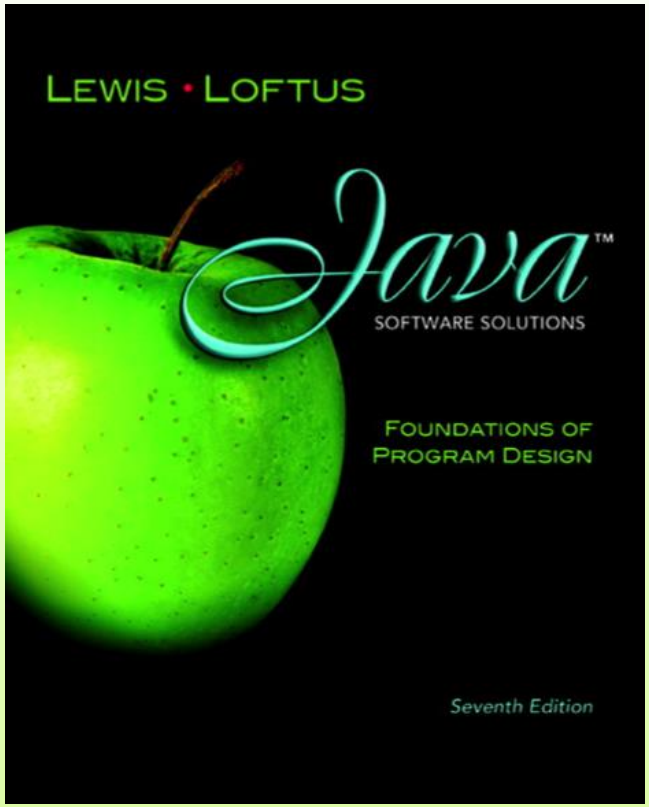
Week 1 & 2
Writing Classes



Java Software Solutions

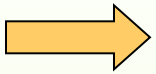
Foundations of Program Design

Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Outline



Creating Objects

The String Class

The Random and Math Classes

Formatting Output

Enumerated Types

Wrapper Classes

Creating Objects

- A variable holds either a primitive value or a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

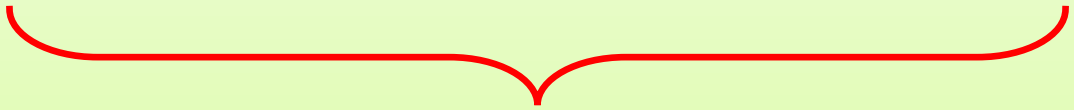
```
String title;
```

- No object is created with this declaration
- An object reference variable holds the address of an object
- The object itself must be created separately

Creating Objects

- Generally, we use the `new` operator to create an object
- Creating an object is called *instantiation*
- An object is an *instance* of a particular class

```
title = new String ("Java Software Solutions");
```



This calls the String *constructor*, which is a special method that sets up the object

Invoking Methods

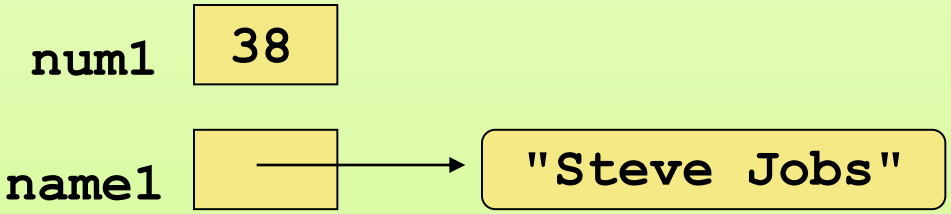
- Once an object has been instantiated, we can use the *dot (.)operator* to invoke its methods

```
numChars = title.length()
```

- A method may *return a value*, which can be used in an assignment or expression
- A method invocation can be thought of as asking an object to perform a service

References

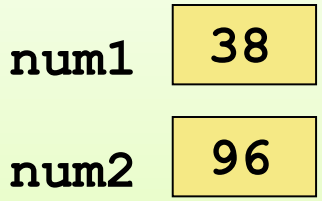
- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically



Assignment Revisited

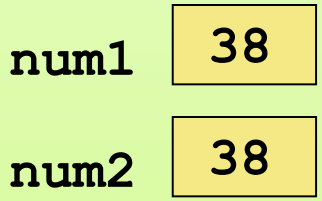
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

Before:




```
num2 = num1;
```

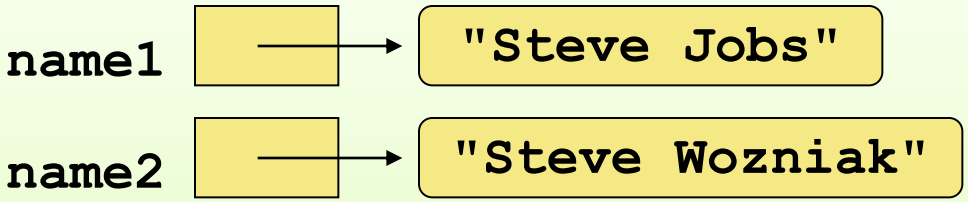
After:



Reference Assignment

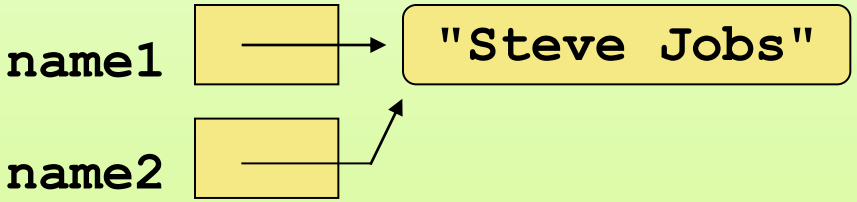
- For object references, assignment copies the address:

Before:




```
name2 = name1;
```

After:



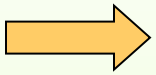
Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- That creates an interesting situation: one object can be accessed using multiple reference variables
- Aliases can be useful, but should be managed carefully
- Changing an object through one reference changes it for all of its aliases, because there is really only one object

Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer is responsible for performing garbage collection

Outline



Creating Objects

The String Class

The Random and Math Classes

Formatting Output

Enumerated Types

Wrapper Classes

The String Class

- Because strings are so common, we don't have to use the `new` operator to create a `String` object

```
title = "Java Software Solutions";
```

- !!!This is a special syntax that works only for strings

- Each string literal (enclosed in double quotes) represents a `String` object

String Methods

- Once a `String` object has been created, neither its value nor its length can be changed
- Therefore we say that an object of the `String` class is *immutable*
- However, several methods of the `String` class return new `String` objects that are modified versions of the original

String Indexes

- It is occasionally helpful to refer to a particular character within a string
- This can be done by specifying the character's numeric *index*
- The indexes begin at zero in each string
- In the string `"Hello"`, the character `'H'` is at index 0 and the `'o'` is at index 4
- See `StringMutation.java` in the next Slide

```
//********************************************************************
//  StringMutation.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the String class and its methods.
//********************************************************************

public class StringMutation
{
   //-----------------------------------------------------------------
   //  Prints a string and various mutations of it.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      String phrase = "Change is inevitable";
      String mutation1, mutation2, mutation3, mutation4;

      System.out.println ("Original string: \"" + phrase + "\"");
      System.out.println ("Length of string: " + phrase.length());

      mutation1 = phrase.concat (", except from vending machines.");
      mutation2 = mutation1.toUpperCase();
      mutation3 = mutation2.replace ('E', 'X');
      mutation4 = mutation3.substring (3, 30);
```

continued

continued

```
        // Print each mutated string
        System.out.println ("Mutation #1: " + mutation1);
        System.out.println ("Mutation #2: " + mutation2);
        System.out.println ("Mutation #3: " + mutation3);
        System.out.println ("Mutation #4: " + mutation4);

        System.out.println ("Mutated length: " + mutation4.length());
    }
}
```

Output

```
Original string: "Change is inevitable"
Length of string: 20
Mutation #1: Change is inevitable, except from vending machines.
Mutation #2: CHANGE IS INEVITABLE, EXCEPT FROM VENDING MACHINES.
Mutation #3: CHANGX IS INXVITABLX, XXCXPT FROM VXNDING MACHINXS.
Mutation #4: NGX IS INXVITABLX, XXCXPT F
Mutated length: 27
```

```
        System.out.println ("Mutated length: " + mutation4.length());
    }
}
```

Quick Check

What output is produced by the following?

```
String str = "Space, the final frontier.";
System.out.println (str.length());
System.out.println (str.substring(7));
System.out.println (str.toUpperCase());
System.out.println (str.length());
```

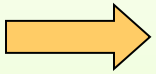
Quick Check

What output is produced by the following?

```
String str = "Space, the final frontier.";
System.out.println (str.length());
System.out.println (str.substring(7));
System.out.println (str.toUpperCase());
System.out.println (str.length());
```

```
26
the final frontier.
SPACE, THE FINAL FRONTIER.
26
```

Outline



Creating Objects

The String Class

The Random and Math Classes

Formatting Output

Enumerated Types

Wrapper Classes

The Random Class

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudorandom numbers
- A `Random` object performs complicated calculations based on a *seed value* to produce a stream of seemingly random values
- See `RandomNumbers.java`

```
//********************************************************************
//  RandomNumbers.java       Author: Lewis/Loftus
//
//  Demonstrates the creation of pseudo-random numbers using the
//  Random class.
//********************************************************************

import java.util.Random;

public class RandomNumbers
{
   //-----------------------------------------------------------------
   //  Generates random numbers in various ranges.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Random generator = new Random();
      int num1;
      float num2;

      num1 = generator.nextInt();
      System.out.println ("A random integer: " + num1);

      num1 = generator.nextInt(10);
      System.out.println ("From 0 to 9: " + num1);
```

continued

continued

```
        num1 = generator.nextInt(10) + 1;
        System.out.println ("From 1 to 10: " + num1);

        num1 = generator.nextInt(15) + 20;
        System.out.println ("From 20 to 34: " + num1);

        num1 = generator.nextInt(20) - 10;
        System.out.println ("From -10 to 9: " + num1);

        num2 = generator.nextFloat();
        System.out.println ("A random float (between 0-1): " + num2);

        num2 = generator.nextFloat() * 6;  // 0.0 to 5.999999
        num1 = (int)num2 + 1;
        System.out.println ("From 1 to 6: " + num1);
    }
}
```

continued

```
        num1
        Syst

        num1
        Syst

        num1
        Syst

        num2 = generator.nextFloat();
        System.out.println ("A random float (between 0-1): " + num2);

        num2 = generator.nextFloat() * 6;  // 0.0 to 5.999999
        num1 = (int)num2 + 1;
        System.out.println ("From 1 to 6: " + num1);
    }
}
```

Sample Run

```
A random integer: 672981683
From 0 to 9: 0
From 1 to 10: 3
From 20 to 34: 30
From -10 to 9: -4
A random float (between 0-1): 0.18538326
From 1 to 6: 3
```

Quick Check

Given a `Random` object named `gen`, what range of values are produced by the following expressions?

```
gen.nextInt(25)
gen.nextInt(6) + 1
gen.nextInt(100) + 10
gen.nextInt(50) + 100
gen.nextInt(10) – 5
gen.nextInt(22) + 12
```

Quick Check

Given a `Random` object named `gen`, what range of values are produced by the following expressions?

	Range
`gen.nextInt(25)`	`0 to 24`
`gen.nextInt(6) + 1`	`1 to 6`
`gen.nextInt(100) + 10`	`10 to 109`
`gen.nextInt(50) + 100`	`100 to 149`
`gen.nextInt(10) – 5`	`-5 to 4`
`gen.nextInt(22) + 12`	`12 to 33`

Quick Check

Write an expression that produces a random integer in the following ranges:

Range
`0 to 12`
`1 to 20`
`15 to 20`
`-10 to 0`

Quick Check

Write an expression that produces a random integer in the following ranges:

Range	
`0 to 12`	`gen.nextInt(13)`
`1 to 20`	`gen.nextInt(20) + 1`
`15 to 20`	`gen.nextInt(6) + 15`
`-10 to 0`	`gen.nextInt(11) - 10`

The Math Class

- The `Math` class is part of the `java.lang` package
- The `Math` class contains methods that perform various mathematical functions
- These include:
  - absolute value
  - square root
  - exponentiation
  - trigonometric functions

The Math Class

- The methods of the `Math` class are *static methods* (also called *class methods*)
- Static methods are invoked through the class name – no object of the `Math` class is needed

```
value = Math.cos(90) + Math.sqrt(delta);
```

- We discuss static methods further in Chapter 7
- See `Quadratic.java`

```
//********************************************************************
//  Quadratic.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the Math class to perform a calculation
//  based on user input.
//********************************************************************

import java.util.Scanner;

public class Quadratic
{
   //-----------------------------------------------------------------
   //  Determines the roots of a quadratic equation.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int a, b, c;  // ax^2 + bx + c
      double discriminant, root1, root2;

      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter the coefficient of x squared: ");
      a = scan.nextInt();
```

continued

continued

```
      System.out.print ("Enter the coefficient of x: ");
      b = scan.nextInt();

      System.out.print ("Enter the constant: ");
      c = scan.nextInt();

      // Use the quadratic formula to compute the roots.
      // Assumes a positive discriminant.

      discriminant = Math.pow(b, 2) - (4 * a * c);
      root1 = ((-1 * b) + Math.sqrt(discriminant)) / (2 * a);
      root2 = ((-1 * b) - Math.sqrt(discriminant)) / (2 * a);

      System.out.println ("Root #1: " + root1);
      System.out.println ("Root #2: " + root2);
   }
}
```

continued

```
        System
        b = sc

        System
        c = sc

        // Use the quadratic formula to compute the roots.
        // Assumes a positive discriminant.

        discriminant = Math.pow(b, 2) - (4 * a * c);
        root1 = ((-1 * b) + Math.sqrt(discriminant)) / (2 * a);
        root2 = ((-1 * b) - Math.sqrt(discriminant)) / (2 * a);

        System.out.println ("Root #1: " + root1);
        System.out.println ("Root #2: " + root2);
    }
}
```

Sample Run

```
Enter the coefficient of x squared: 3
Enter the coefficient of x: 8
Enter the constant: 4
Root #1: -0.6666666666666666
Root #2: -2.0
```

Outline

Creating Objects

The String Class

The Random and Math Classes

Formatting Output

Enumerated Types

Wrapper Classes

Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Wrapper Classes

- The following declaration creates an `Integer` object which represents the integer 40 as an object

```
Integer age = new Integer(40);
```

- An object of a wrapper class can be used in any situation where a primitive value will not suffice
- For example, some objects serve as containers of other objects
- Primitive values could not be stored in such containers, but wrapper objects could be

Wrapper Classes

- Wrapper classes also contain static methods that help manage the associated type
- For example, the `Integer` class contains a method to convert an integer stored in a `String` to an `int` value:

```
num = Integer.parseInt(str);
```

- They often contain useful constants as well
- For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest `int` values

Autoboxing

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;
int num = 42;
obj = num;
```

- The assignment creates the appropriate `Integer` object
- The reverse conversion (called *unboxing*) also occurs automatically as needed

Quick Check

Are the following assignments valid? Explain.

```
Double value = 15.75;
```

```
Character ch = new Character('T');
char myChar = ch;
```

Quick Check

Are the following assignments valid? Explain.

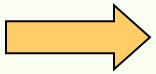
```
Double value = 15.75;
```

Yes. The double literal is autoboxed into a `Double` object.

```
Character ch = new Character('T');
char myChar = ch;
```

Yes, the char in the object is unboxed before the assignment.

Outline



Anatomy of a Class

Encapsulation

Anatomy of a Method

Writing Classes

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

Examples of Classes

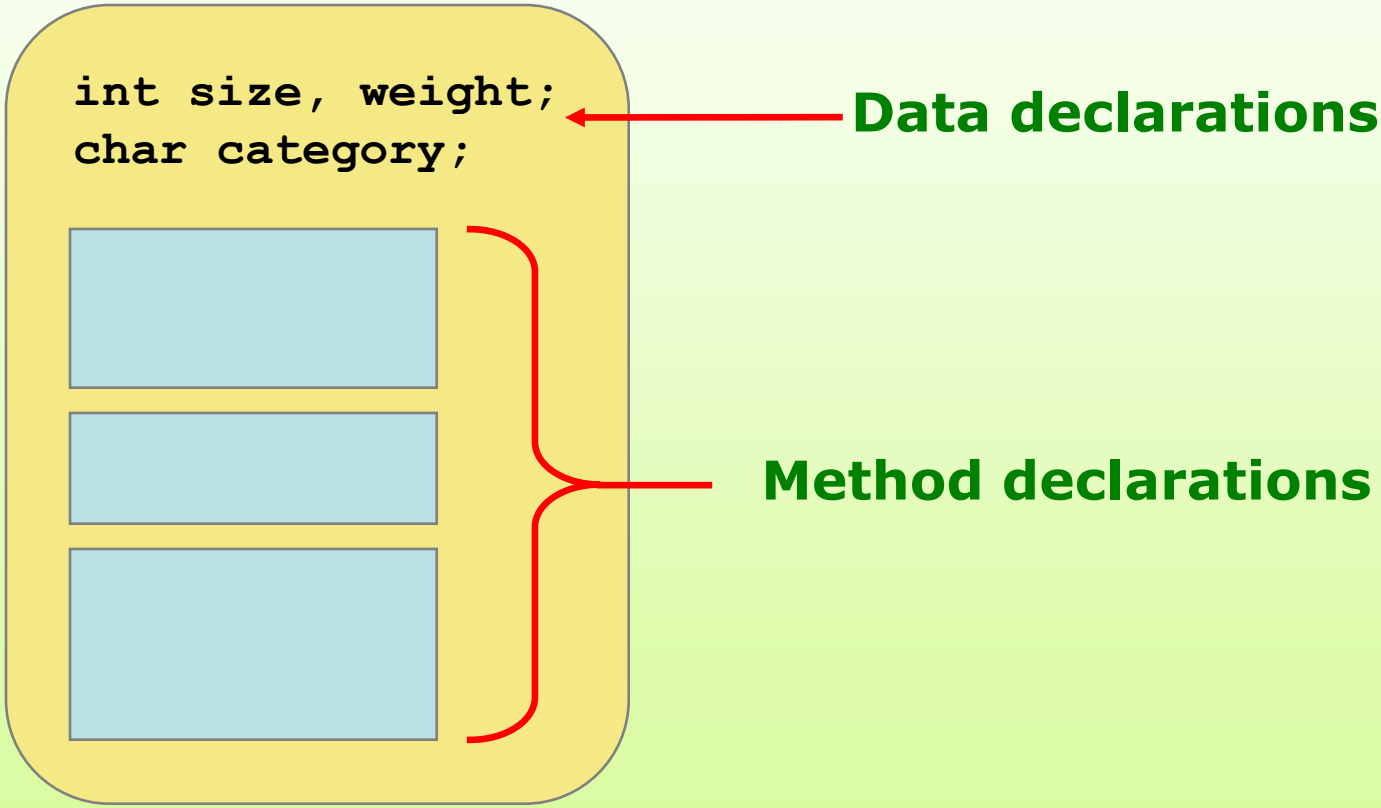
Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight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

Classes and Objects

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

Classes

- A class can contain data declarations and method declarations



Classes

- The values of the data define the state of an object created from the class
- The functionality of the methods define the behaviors of the object

For our `Die` class, we might declare an integer called `faceValue` that represents the current value showing on the face

One of the methods would "roll" the die by setting `faceValue` to a random number between one and six

Classes

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

```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) 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;
   }
```

continue

continue

```
//-----------------------------------------------------------------
//  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;
}
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Returns a string representation of this die.
    //-----------------------------------------------------------------
    public String toString()
    {
        String result = Integer.toString(faceValue);

        return result;
    }
}
```

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

continue

continue

```
        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);
    }
}
```

continue

```
        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
```

The Die Class

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

The toString Method

- It's good practice to define a `toString` method for a class
- The `toString` method returns a character string that represents the object in some way

`toString` method is called automatically when an object is concatenated to a string or when it is passed to the `println` method

Constructors

- As mentioned previously, a *constructor* is used to set up an object when it is initially created
- A constructor has the same name as the class

The `Die` constructor is used to set the initial face value of each new die object to one

We examine constructors in more detail later in this chapter

Data Scope

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

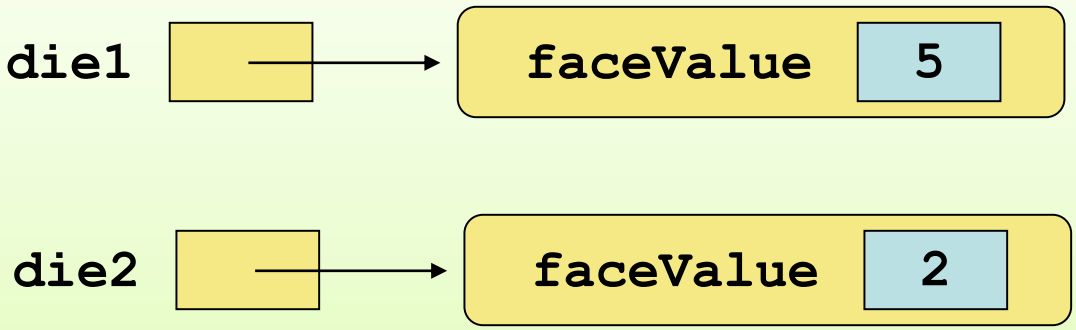
In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else

Instance Data

- A variable declared at the class level (such as `faceValue`) is called *instance data*
  - Each object of the class has its own instance variable
- Each time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space (! they dont share the instance data !)
- That is, two objects can have different states

Instance Data

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



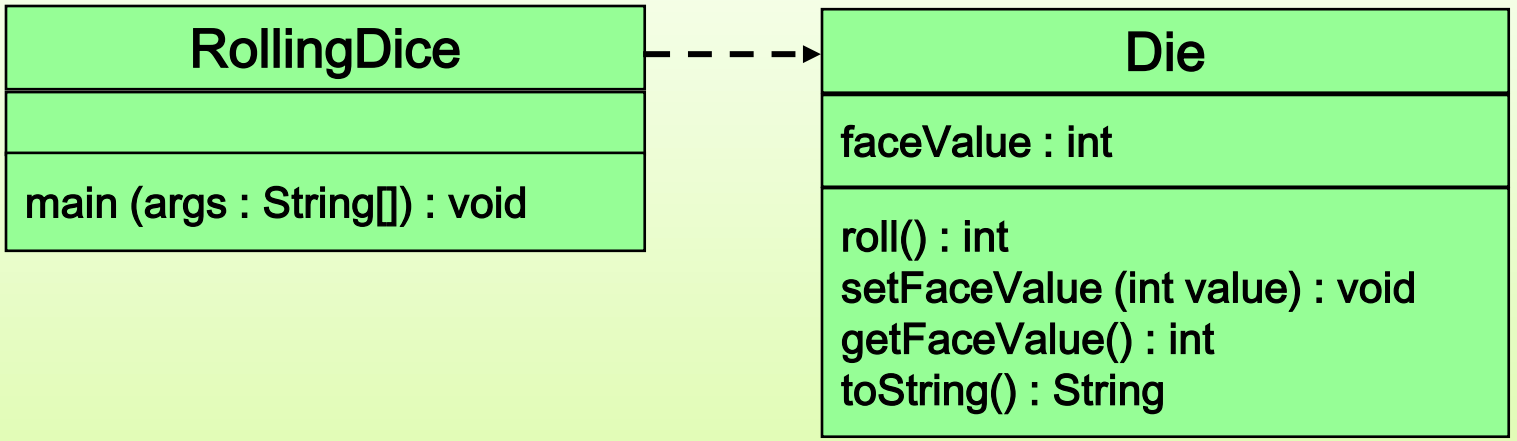
Each object maintains its own `faceValue` variable, and thus has its own state

UML Diagrams

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

UML Class Diagrams

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



Quick Check

What is the relationship between a class and an object?

Quick Check

What is the relationship between a class and an object?

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

Quick Check

Where is instance data declared?

What is the scope of instance data?

What is local data?

Quick Check

Where is instance data declared?

At the class level.

What is the scope of instance data?

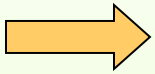
It can be referenced in any method of the class.

What is local data?

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

Outline

Anatomy of a Class



Encapsulation

Anatomy of a Method

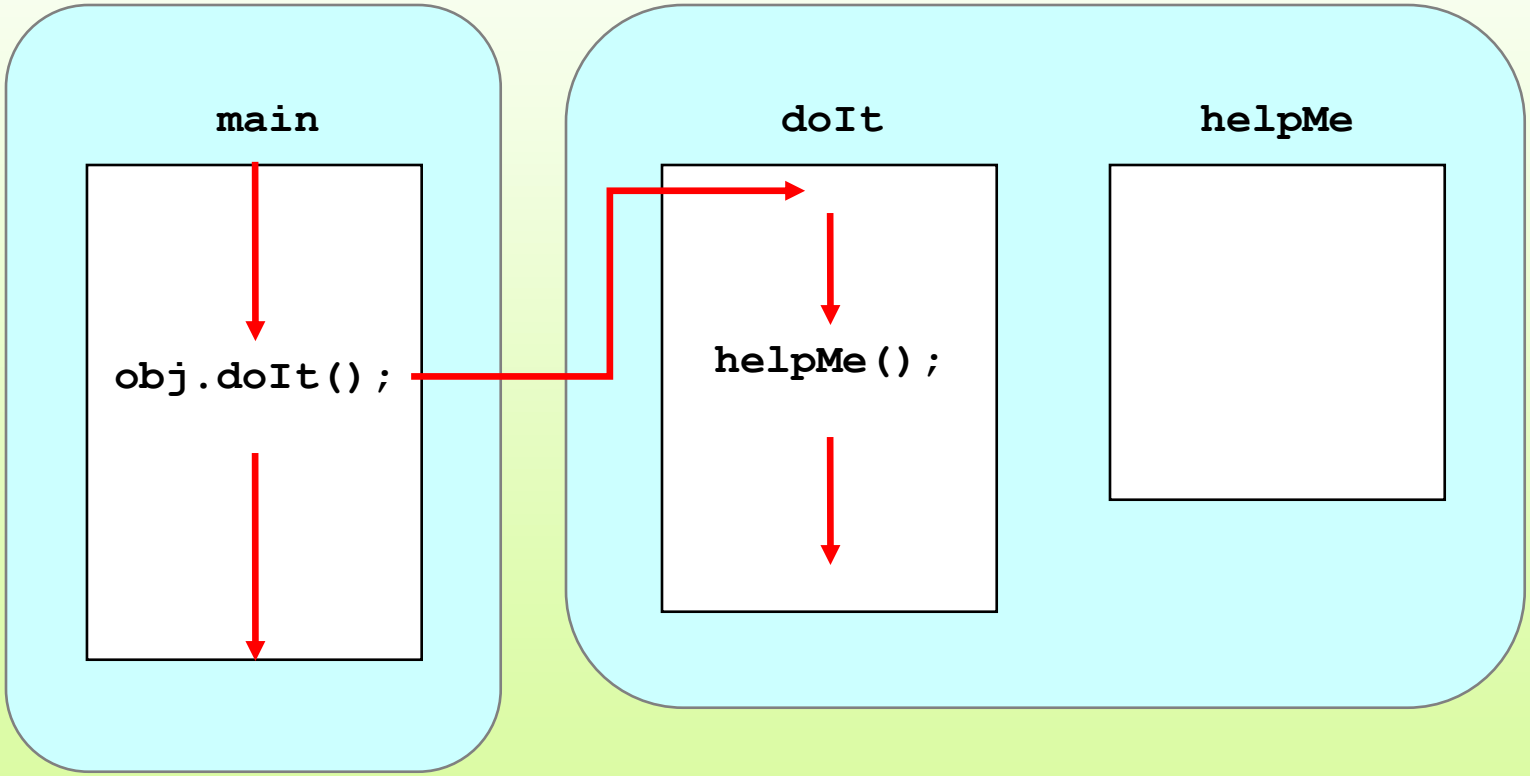
Encapsulation

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

Encapsulation

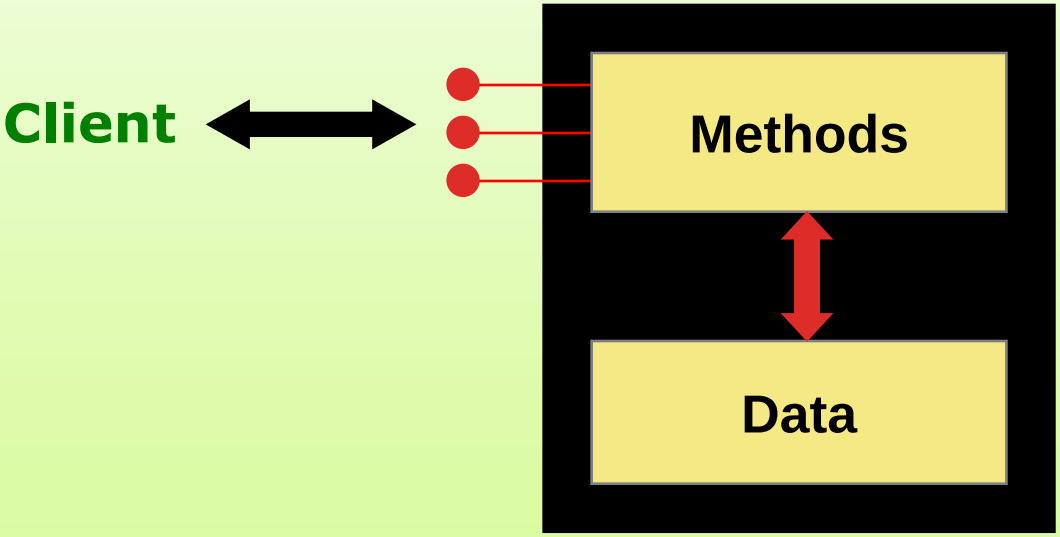
- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (i.e.call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made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

Encapsulation



Encapsulation

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



Visibility Modifiers

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

Visibility Modifiers

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

Visibility Modifiers

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

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support method*
- Since a support method is not intended to be called by any client, it should not be declared with public visibility

Visibility Modifiers

	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients**	**Support other methods in the class**

Encapsulation and Visibility Modifiers

```
class MyRectangle {
    private double wid
    private doubl
    private Point leftTop;

    public double getWidth(){ return width; }
    public void setWidth(double width){this.width=width;}
    public void translate(double deltaX, double deltaY){…}
}
```

Implementation details hidden

Proper class interface

```
class MyRectangle {
    private Rectangle rect;

    public double getWidth(){ return rect.getWidth(); }
    public void setWidth(double width){
        rect.setSize(width, rect.getHeigh
    }
    public void translate(double deltaX, double deltaY){…}
}
```

Implementation changed

Class interface unchanged

Accessors and Mutators

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

Mutator Restrictions

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

Quick Check

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

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

Quick Check

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

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

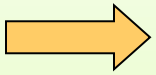
Why is it ok to declare `MAX` as `public` in the `Die` class?

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

Outline

Anatomy of a Class

Encapsulation



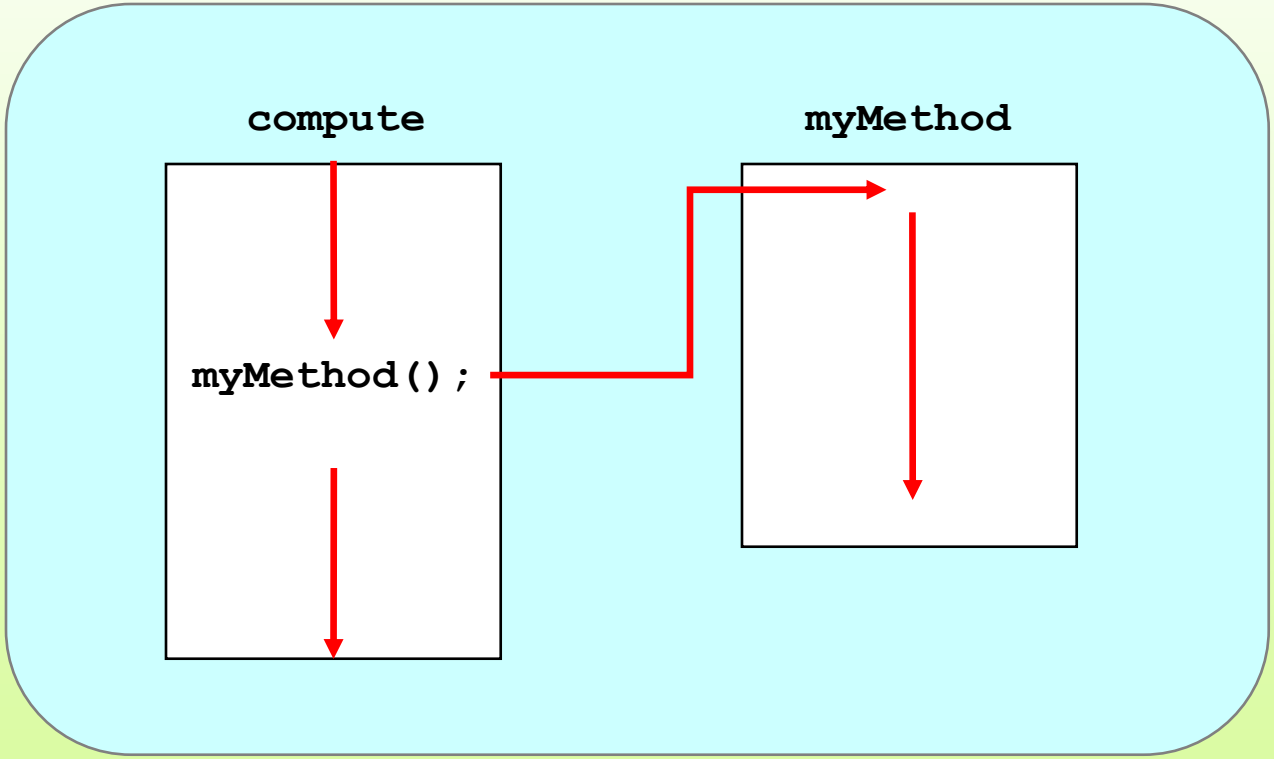
Anatomy of a Method

Method Declarations

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

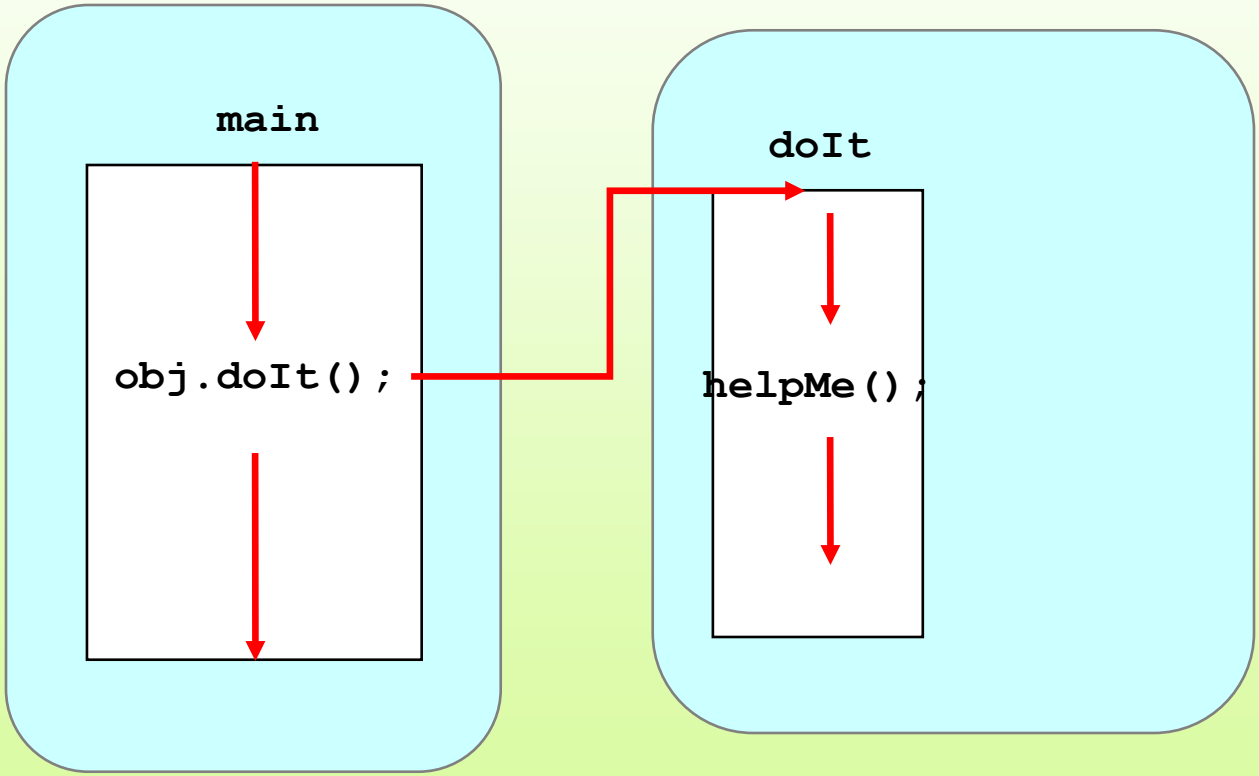
Method Control Flow

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



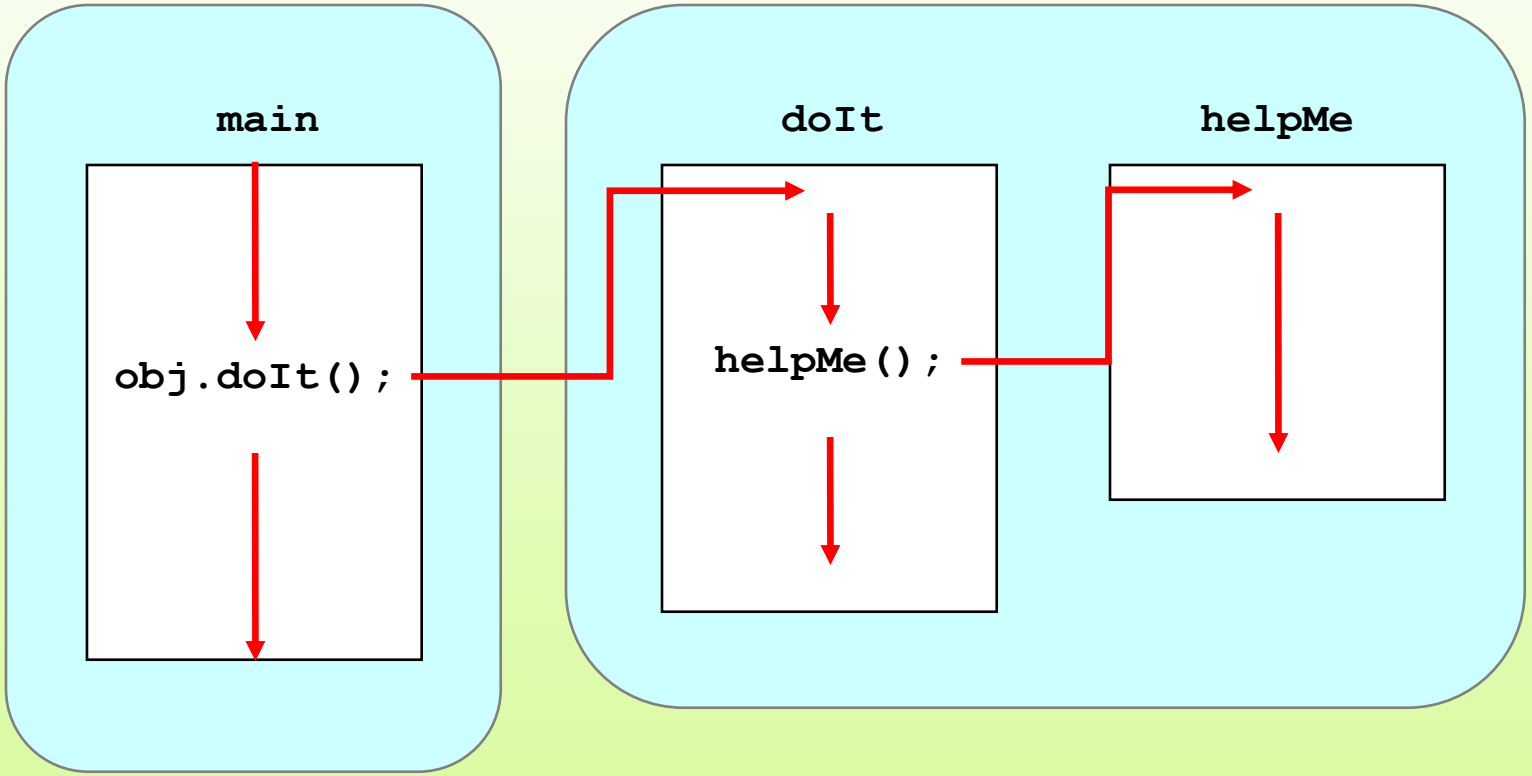
Method Control Flow

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



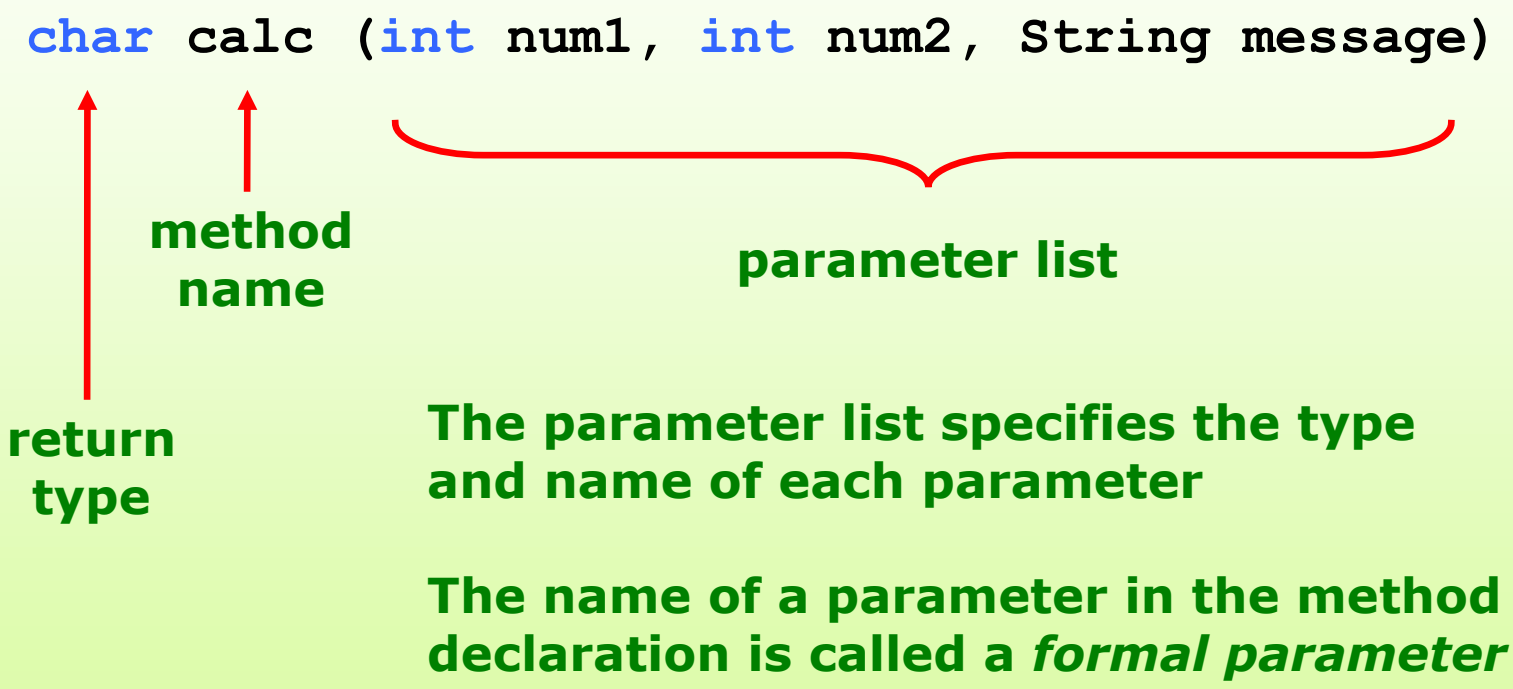
Method Control Flow

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



Method Header

- A method declaration begins with a *method header*



Method Body

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

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

   return result;
}
```

The return expression must be consistent with the return type

`sum` **and** `result` **are local data**

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

The return Statement

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

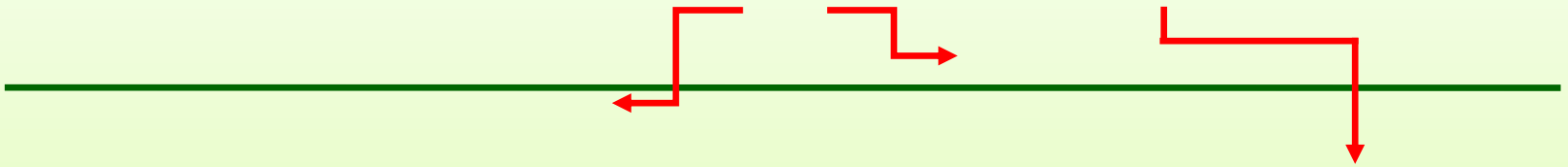
```
return expression;
```

- Its expression must conform to the return type

Parameters

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

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



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

   return result;
}
```

Local Data

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

Bank Account Example

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

Driver Programs

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

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

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

continue

continue

```
      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);
   }
}
```

continue

```
        System.                                                    " +


        acct1.a
        acct2.a
        acct3.a

        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;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  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;
   }
```

continue

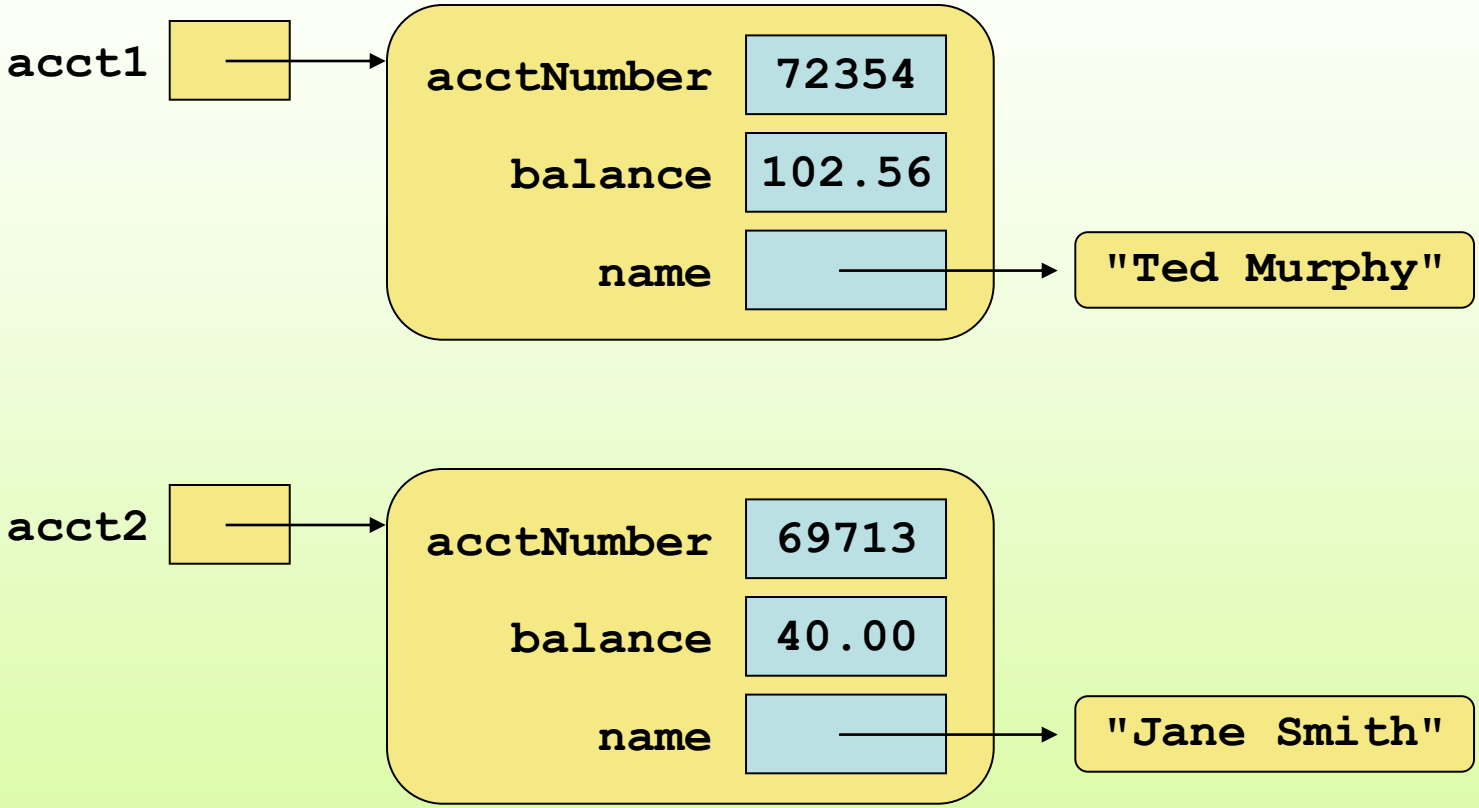
continue

```
    //-----------------------------------------------------------------
    //  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));
    }
}
```

Bank Account Example



Bank Account Example

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

Constructors Revisited

- Note that a constructor has no return type specified in the method header, not even `void`
- A common error is to put a return type on a constructor, which makes it a “regular” method that happens to have the same name as the class
- The programmer does not have to define a constructor for a class
- 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);
```