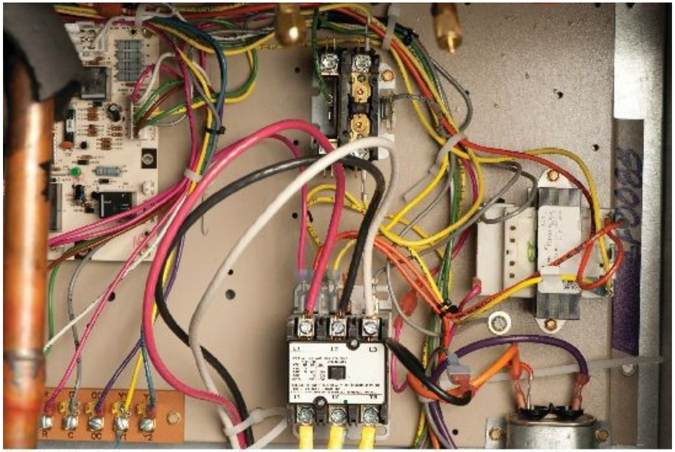
Week 5

Objects & Classes

(Chapter 3)

Chapter Goals



© Kris Hanke/iStockphoto.

- To become familiar with the process of implementing classes
- To be able to implement and test simple methods
- To understand the purpose and use of constructors
- To understand how to access instance variables and local variables
- To be able to write javadoc comments
- To implement classes for drawing graphical shapes

Objects and Classes

© Luc Meaille/iStockphoto.

Each part a home builder uses, such as a furnace or a water heater, fulfills a particular function. Similarly, you build programs from objects, each of which has a particular behavior.

- In Java, you build programs for *objects.*
- Each object has certain behaviors.
- You can manipulate the object to get certain effects.

Using Objects

- Object: an entity in your program that you can manipulate by calling one or more of its methods.
- Method: consists of a sequence of instructions that can access the data of an object.
  - You do not know what the instructions are
  - You do know that the behavior is well defined
- `System.out` has a `println` method
  - You do not know how it works
  - What is important is that it does the work you request of it

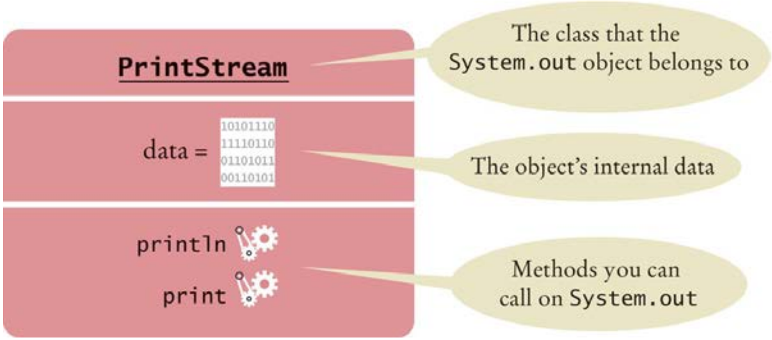

PrintStream
data =
10101110
11110110
01101011
00110101
println
print
The class that the System.out object belongs to
The object's internal data
Methods you can call on System.out

Figure 1 Representation of the `System.out` Object

Classes

- A class describes a set of objects with the same behavior.
- Some string objects

```
"Hello World"
"Goodbye"
"Mississippi"
```

- You can invoke the same methods on all strings.
- `System.out` is a member of the `PrintStream` class that writes to the console window.
- You can construct other objects of `PrintStream` class that write to different destinations.
- All `PrintStream` objects have methods `println` and `print`.

Instance Variables and Encapsulation

© Jasmin Awad/iStockphoto.

Figure 1 Tally counter

- Simulator statements:

```
Counter tally = new
Counter();
tally.click();
tally.click(); // Sets result to 2
int result = tally.getValue(); // Gets result ~ 2
```

- Each counter needs to store a variable that keeps track of the number of simulated button clicks.

Instance Variables

- **Instance variables** store the data of an object.
- **Instance of a class:** an object of the class.
- An instance variable is a storage location present in each object of the class.
- The class declaration specifies the instance variables:

```
public class Counter
{
   private int value;
   ...
}
```

- An object's instance variables store the data required for executing its methods.

Instance Variables

- An instance variable declaration consists of the following parts:
  - access specifier (`private`)
  - type of variable (such as `int`)
  - name of variable (such as `value`)
- You should declare all instance variables as private.

Instance Variables

- Each object of a class has its own set of instance variables.

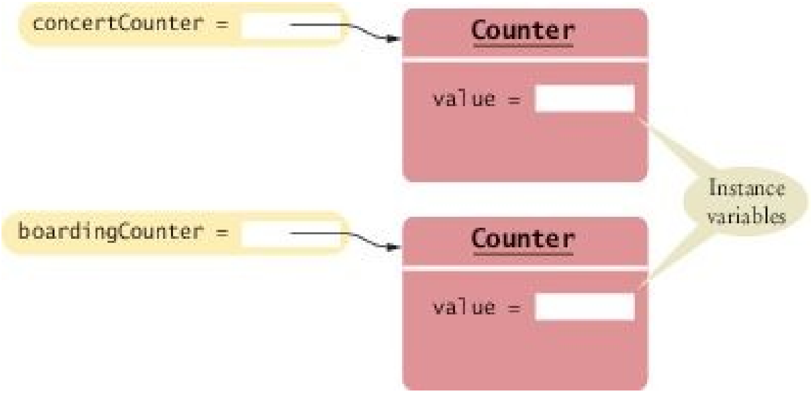
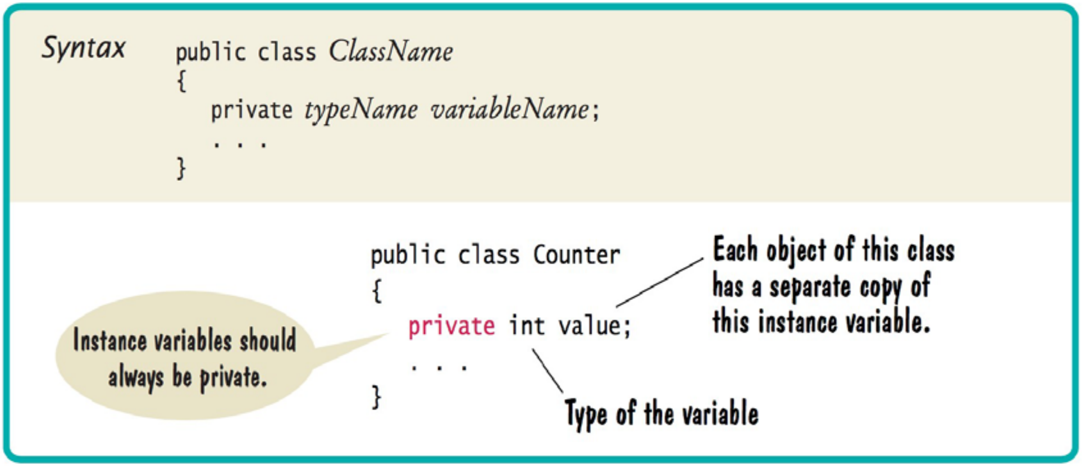


Figure 2 Instance Variables

Syntax 3.1 Instance Variable Declaration



Syntax

```
public class ClassName
{
   private typeName variableName;
   . . .
}
```

```
public class Counter
{
   private int value;
   . . .
}
```

Instance variables should always be private.

Each object of this class has a separate copy of this instance variable.

Type of the variable

Instance Variables

These clocks have common behavior, but each of them has a different state. Similarly, objects of a class can have their instance variables set to different values.

© Mark Evans/iStockphoto.

The Methods of the Counter Class

- The `click` method advances the counter value by 1:

```
public void click()
{
   value = value + 1;
}
```

  - Affects the value of the instance variable of the object on which the method is invoked
  - The method call `concertCounter.click();`
    - Advances the value variable of the concertCounter object

The Methods of the Counter Class

- The `getValue` method returns the current value:

```
public int getValue()
{
   return value;
}
```

  - The return statement
    - Terminates the method call
    - Returns a result (the return value) to the method's caller
  - Private instance variables can only be accessed by methods of the same class.

Encapsulation

- **Encapsulation** is the process of hiding implementation details and providing methods for data access.
- To encapsulate data:
  - Declare instance variables as `private` and
  - Declare public methods that access the variables
- Encapsulation allows a programmer to use a class without having to know its implementation.
- Information hiding makes it simpler for the implementor of a class to locate errors and change implementations.

Encapsulation

© yenwen/iStockphoto.

A thermostat functions as a "black box" whose inner workings are hidden.

- When you assemble classes, like `Rectangle` and `String`, into programs you are like a contractor installing a thermostat.
- When you implement your own classes you are like the manufacturer who puts together a thermostat out of parts.

section_1/Counter.java

```java
/**
This class models a tally counter.
*/
public class Counter
{
private int value;

/**
Gets the current value of this counter.
@return the current value
*/
public int getValue()
{
return value;
}

/**
Advances the value of this counter by 1.
*/
public void click()
{
value = value + 1;
}

/**
Resets the value of this counter to 0.
*/
public void reset()
{
value = 0;
}
}
```

Specifying the Public Interface of a Class

- In order to implement a class, you first need to know which methods are required.
- Essential behavior of a bank account:
  - deposit money
  - withdraw money
  - get balance

Specifying the Public Interface of a Class

- We want to support method calls such as the following:

```
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```

- Here are the method headers needed for a `BankAccount` class:

```
public void deposit(double amount)

public void withdraw(double amount)

public double getBalance()
```

Specifying the Public Interface of a Class: Method Declaration

- A method's *body* consisting of statements that are executed when the method is called:

```
public void deposit(double amount)
{
   implementation - filled in later
}
```

- You can fill in the method body so it compiles:

```
public double getBalance()
{
   // TODO: fill in
   implementation  return 0;
}
```

Specifying the Public Interface of a Class

- `BankAccount` methods were declared as `public`.
- `public` methods can be called by all other methods in the program.
- Methods can also be declared `private`
  - `private` methods only be called by other methods in the same class
  - `private` methods are not part of the public interface

Specifying Constructors

- Initialize objects
- Set the initial data for objects
- Similar to a method with two differences:
  - The name of the constructor is always the same as the name of the class
  - Constructors have no return type

Specifying Constructors: BankAccount

- Two constructors

```
public BankAccount()
public BankAccount(double initialBalance)
```

- Usage

```
BankAccount harrysChecking = new BankAccount();
BankAccount momsSavings = new BankAccount(5000);
```

Specifying Constructors: BankAccount

- The constructor name is always the same as the class name.
- The compiler can tell them apart because they take different arguments.
- A constructor that takes no arguments is called a no-argument constructor.
- `BankAccount`'s no-argument constructor - header and body:

```
public BankAccount()
{
   constructor body—implementation filled in later
}
```

- The statements in the constructor body will set the instance variables of the object.

BankAccount Public Interface

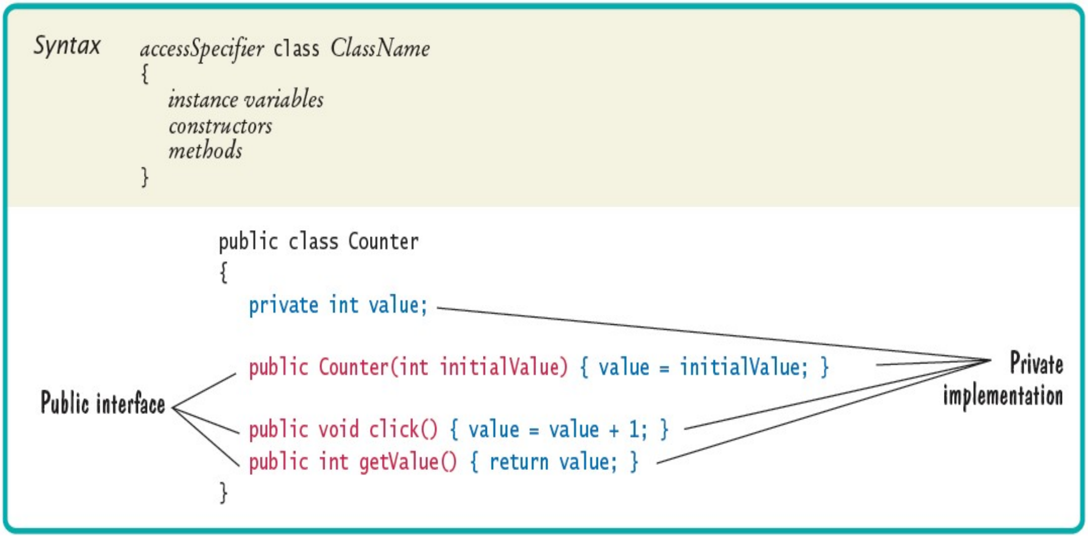
The constructors and methods of a class go inside the class declaration:

```
public class BankAccount
{
   // private instance variables--filled in  later

   // Constructors
   public BankAccount()
   {
      // body--filled in later
   }
   public BankAccount(double initialBalance)
   {
      // body--filled in later
   }

   // Methods
   public void deposit(double amount)
   {
      // body--filled in later
   }
   public void withdraw(double amount)
   {
      // body--filled in later
   }
   public double getBalance()
   {
      // body--filled in later
   }
}
```

Specifying the Public Interface of a Class

- public constructors and methods of a class form the **public interface** of the class.
- These are the operations that any programmer can use

Syntax 3.2 Class Declaration



Syntax

```
accessSpecifier class ClassName
{
   instance variables
   constructors
   methods
}
```

```
public class Counter
{
   private int value;

   public Counter(int initialValue) { value = initialValue; }

   public void click() { value = value + 1; }
   public int getValue() { return value; }
}
```

Using the Public Interface

- Example: transfer money

```
// Transfer from one account to another
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount)
```

- Example: add interest

```
double interestRate = 5; // 5 percent interest
double interestAmount = momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

- Programmers use objects of the `BankAccount` class to carry out meaningful tasks
  - without knowing how the `BankAccount` objects store their data
  - without knowing how the `BankAccount` methods do their work

Commenting the Public Interface

- Use documentation comments to describe the classes and public methods of your programs.
- Java has a standard form for documentation comments.
- A program called `javadoc` can automatically generate a set of HTML pages.
- Documentation comment

  placed before the class or method declaration that is being documented

Commenting the Public Interface - Documenting a method

- Start the comment with a `/**`.
- Describe the method’s purpose.
- Describe each parameter:
  - start with `@param`
  - name of the parameter that holds the argument
  - a short explanation of the argument
- Describe the return value:
  - start with `@return`
  - describe the return value
- Omit `@param` tag for methods that have no arguments.
- Omit the `@return` tag for methods whose return type is `void`.
- End with */

Commenting the Public Interface - Documenting a method

- Example:

```
/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
   implementation—filled in later
}
```

- Example:

```
/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   implementation—filled in later
}
```

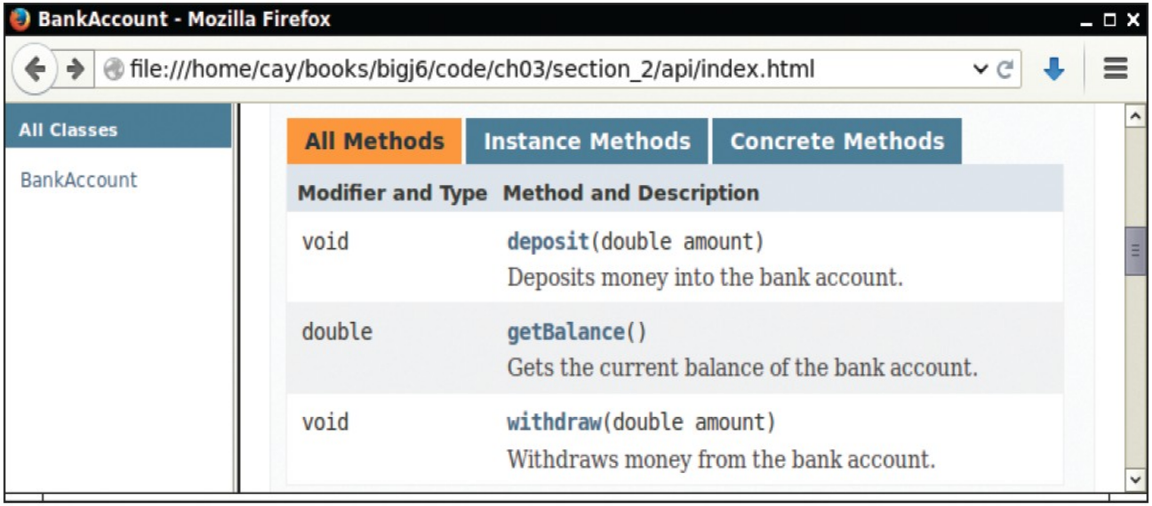
Commenting the Public Interface - Documenting a class

- Place above the class declaration.
- Supply a brief comment explaining the class's purpose.
- Example:

  ```
  /**
     A bank account has a balance that can be changed by
     deposits and withdrawals.
  */
  public class BankAccount
  {
     . . .
  }
  ```

  - Provide documentation comments for:
    - every class
    - every method
    - every parameter variable
    - every return value

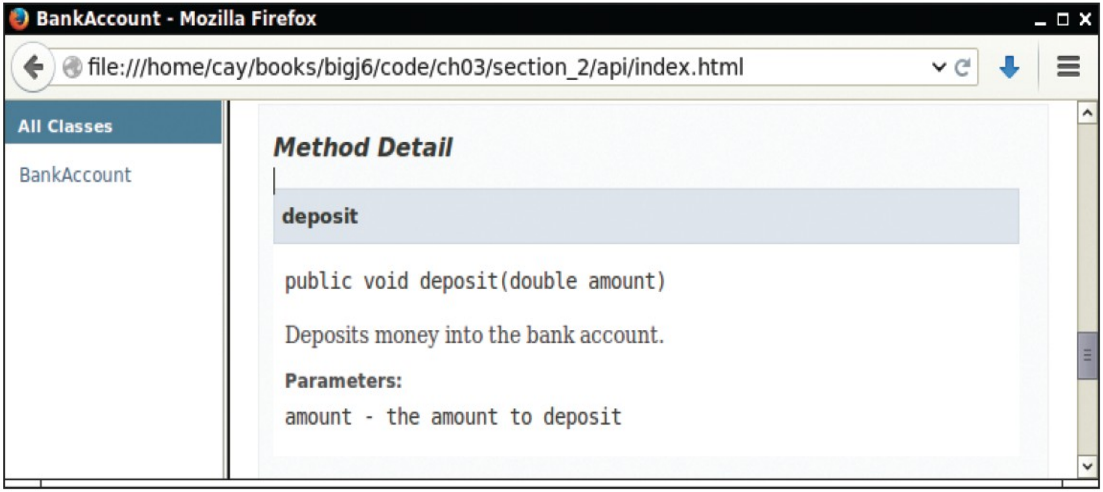
Method Summary


BankAccount - Mozilla Firefox
file:///home/cay/books/bigj6/code/ch03/section_2/api/index.html
All Classes
BankAccount
All Methods
Instance Methods
Concrete Methods

Modifier and Type	Method and Description
void	deposit(double amount) Deposits money into the bank account.
double	getBalance() Gets the current balance of the bank account.
void	withdraw(double amount) Withdraws money from the bank account.

Figure 3 A Method Summary Generated by `javadoc`

Method Details



BankAccount - Mozilla Firefox
file:///home/cay/books/bigj6/code/ch03/section_2/api/index.html
All Classes
BankAccount

Method Detail

deposit

```
public void deposit(double amount)
```

Deposits money into the bank account.

Parameters:

`amount - the amount to deposit`

Figure 4 Method Detail Generated by `javadoc`

Providing the Class Implementation

- The private implementation of a class consists of:

  instance variables

  the bodies of constructors

  the bodies of methods.

Providing Instance Variables

- Determine the data that each bank account object contains.
- What does the object need to remember so that it can carry out its methods?
- Each bank account object only needs to store the current balance.
- `BankAccount` instance variable declaration:

```
public class BankAccount
{
   private double balance;
   // Methods and constructors below
   . . .
}
```

Providing Instance Variables

Like a wilderness explorer who needs to carry all items that may be needed, an object needs to store the data required for its method calls.

© iStockphoto.com/migin.

Providing Constructors

- Constructor's job is to initialize the instance variables of the object.
- The no-argument constructor sets the balance to zero.

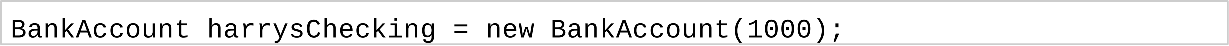
```
public BankAccount()
{
   balance = 0;
}
```

- The second constructor sets the balance to the value supplied as the construction argument.

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

Providing Constructors -Tracing the Statement

Steps carried out when the following statement is executed:



```
BankAccount harrysChecking = new BankAccount(1000);
```

- Create a new object of type `BankAccount`.❶
- Call the second constructor

  because an argument is supplied in the constructor call
- Set the parameter variable `initialBalance` to 1000.❷
- Set the `balance` instance variable of the newly created object to
- `initialBalance`.❸

- Return an object reference, that is, the memory location of the object.
- Store that object reference in the `harrysChecking` variable. ❹

Providing Constructors -Tracing the Statement

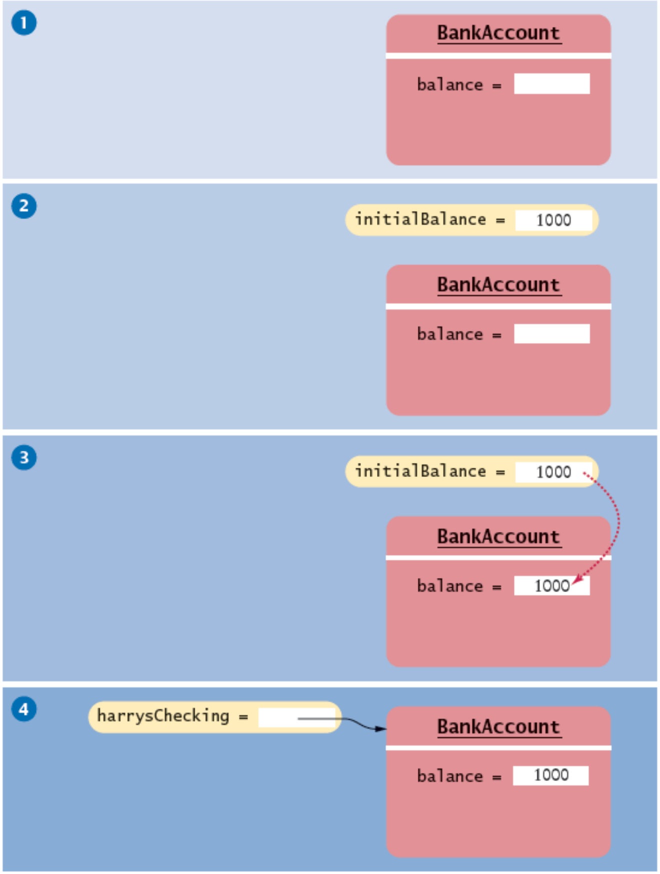


Figure 5 How a Constructor Works

Providing Constructors

© Ann Marie Kurtz/iStockphoto.

A constructor is like a set of assembly instructions for an object.

Providing Methods

- Is the method an accessor or a mutator
  - Mutator method
    - Update the instance variables in some way
  - Accessor method
    - Retrieves or computes a result
- `deposit` method - a mutator method
  - Updates the balance

```
public void deposit(double amount)
{
 balance = balance + amount;
}
```

Providing Methods -continued

- `withdraw` method - another mutator

```
public void withdraw(double amount)
{
   balance = balance - amount;
}
```

- `getBalance` method - an accessor method

  Returns a value

```
public double getBalance()
{
   return balance;
}
```

Table 1 Implementing Classes

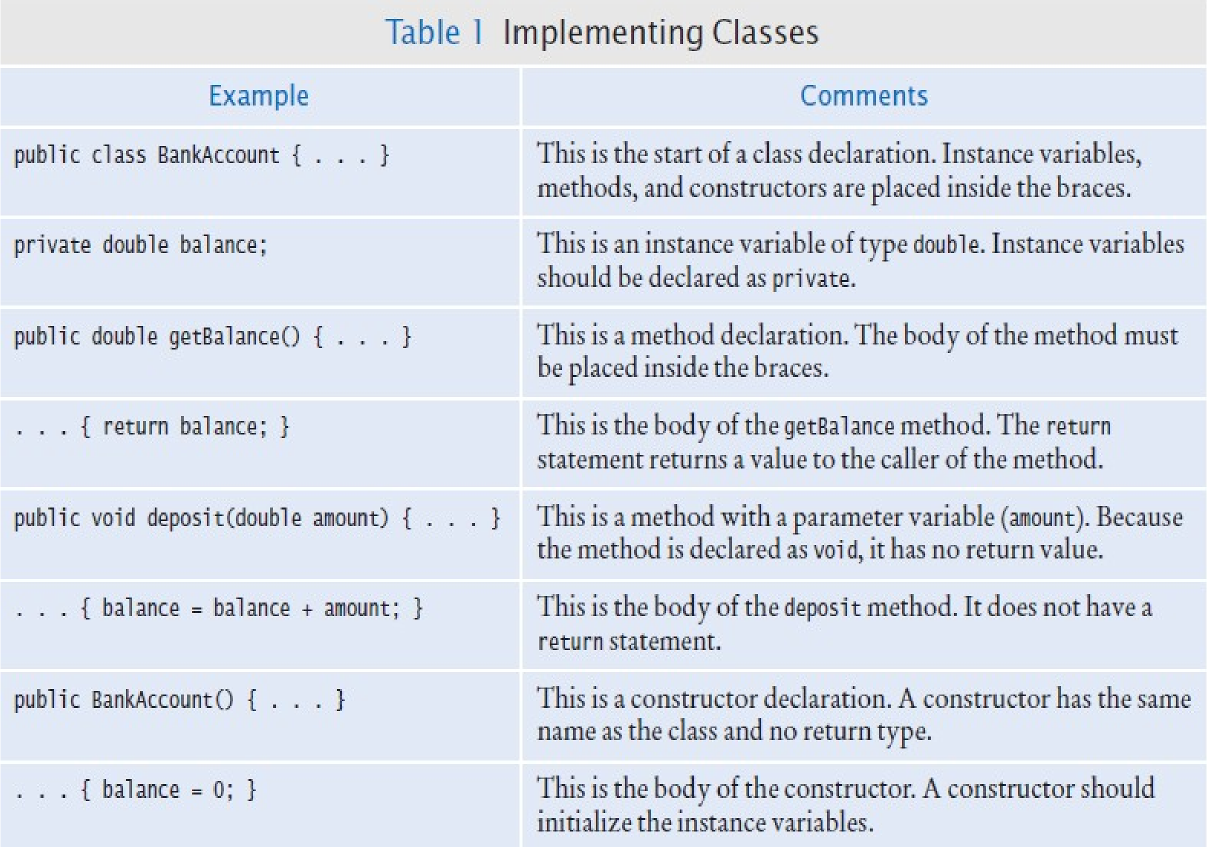


Table 1 Implementing Classes

Example	Comments
`public class BankAccount { . . . }`	This is the start of a class declaration. Instance variables, methods, and constructors are placed inside the braces.
`private double balance;`	This is an instance variable of type `double`. Instance variables should be declared as `private`.
`public double getBalance() { . . . }`	This is a method declaration. The body of the method must be placed inside the braces.
`. . . { return balance; }`	This is the body of the `getBalance` method. The `return` statement returns a value to the caller of the method.
`public void deposit(double amount) { . . . }`	This is a method with a parameter variable (`amount`). Because the method is declared as `void`, it has no return value.
`. . . { balance = balance + amount; }`	This is the body of the `deposit` method. It does not have a `return` statement.
`public BankAccount() { . . . }`	This is a constructor declaration. A constructor has the same name as the class and no return type.
`. . . { balance = 0; }`	This is the body of the constructor. A constructor should initialize the instance variables.

section_3/BankAccount.java

```java
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;

/**
 A car shape that can be positioned anywhere on the screen.
*/
public class Car
{
 private int xLeft;
 private int yTop;

 /**
  Constructs a car with a given top left corner.
  @param x the x coordinate of the top left corner
  @param y the y coordinate of the top left corner
 */
 public Car(int x, int y)
 {
  xLeft = x;
  yTop = y;
 }

 /**
  Draws the car.
  @param g2 the graphics context
 */
 public void draw(Graphics2D g2)
 {
        Rectangle body = new Rectangle(xLeft, yTop + 10,60, 10);
  Ellipse2D.Double frontTire
           = new Ellipse2D.Double(xLeft + 10, yTop + 20,10, 10);
  Ellipse2D.Double rearTire
```

Unit Testing

- `BankAccount.java` can not be executed:
  - It has no main method
  - Most classes do not have a main method
- Before using `BankAccount.java` in a larger program:
  - You should test in isolation
- *Unit test*: verifies that a class works correctly in isolation, outside a complete program.

Unit Testing

- To test a class, either

  use an environment for interactive testing, or

  write a tester class to execute test instructions.

- *Tester class*: a class with a main method that contains statements to test another class.

- Typically carries out the following steps:

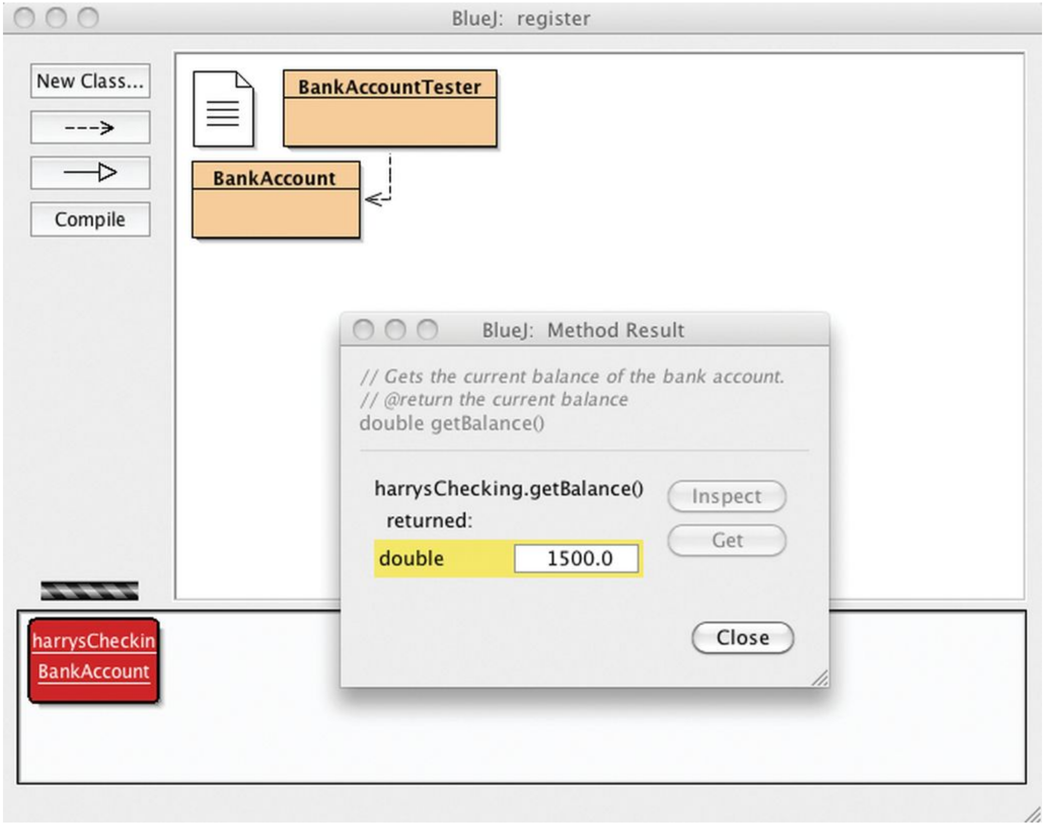
  1. Construct one or more objects of the class that is being tested
  2. Invoke one or more methods
  3. Print out one or more results
  4. Print the expected results

Unit Testing

© Chris Fertnig/iStockphoto.

An engineer tests a part in isolation.
This is an example of unit testing.

Unit Testing with Bluej



```
// Gets the current balance of the bank account.
// @return the current balance
double getBalance()
```

harrysChecking.getBalance() returned:

double 1500.0

Figure 6 The Return Value of the `getBalance` Method in BlueJ

section_4/BankAccountTester.java

```java
/**
   A class to test the BankAccount class.
*/
public class BankAccountTester
{
   /**
      Tests the methods of the BankAccount class.
      @param args not used
   */
   public static void main(String[] args)
   {
      BankAccount harrysChecking = new BankAccount();
      harrysChecking.deposit(2000);
      harrysChecking.withdraw(500);
      System.out.println(harrysChecking.getBalance());
      System.out.println("Expected: 1500");
   }
}
```

Program Run:

```
1500
Expected: 1500
```

Unit Testing - Building a program

- To produce a program: combine both `BankAccount` and `BankAccountTester` classes.
- Details for building the program vary.
- In most environments, you need to carry out these steps:

  1. Make a new subfolder for your program
  2. Make two files, one for each class
  3. Compile both files
  4. Run the test program

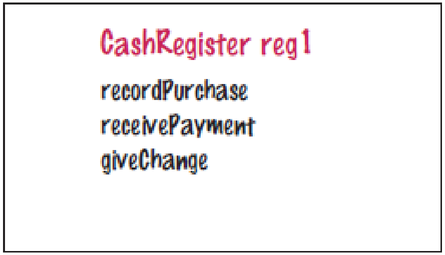
- `BankAccount` and `BankAccountTest` have entirely different purposes:

  `BankAccount` class describes objects that compute bank balances

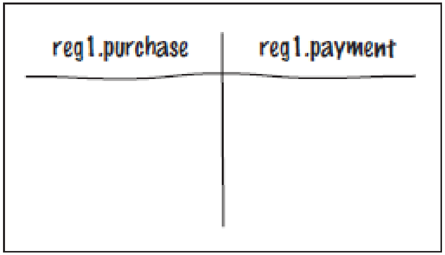
  `BankAccountTester` class runs tests that put a `BankAccount` object through its paces

Problem Solving: Tracing Objects

- Important skill: the ability to simulate the actions of a program with pencil and paper.
- Use an index card or a sticky note for each object:
  - Write the methods on the front
  - Make a table for the values of the instance variables on the back
- `CashRegister` class



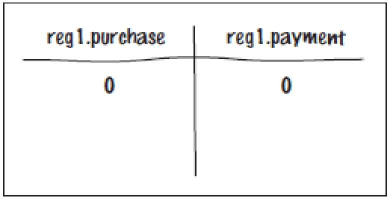
front



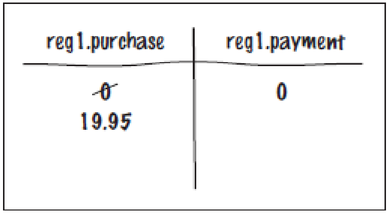
back

Problem Solving: Tracing Objects

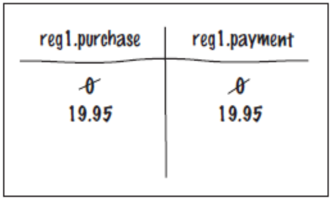
- When an object is constructed, fill in the initial values of the instance variables.
- Update the values of the instance variables when a mutator method is called.
- After a call to `cashRegister's` `recordPurchase` method
- More than one object: create multiple cards



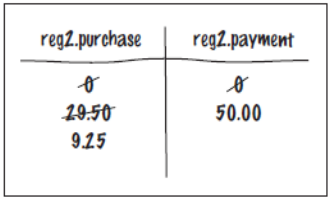
reg1.purchase	reg1.payment
0	0



reg1.purchase	reg1.payment
~~0~~ 19.95	0



reg1.purchase	reg1.payment
~~0~~ 19.95	~~0~~ 19.95



reg2.purchase	reg2.payment
~~0~~ ~~29.50~~ 9.25	~~0~~ 50.00

Problem Solving: Tracing Objects

- Useful when enhancing a class..
- Enhance `CashRegister` class to compute the sales tax.
- Add methods `recordTaxablePurchase` and `getSalesTax` to the front of the card.
- Don't have enough information to compute sales tax:

  need tax rate

  need total of the taxable items

- Need additional instance variables for:

  taxRate

  taxablePurchase

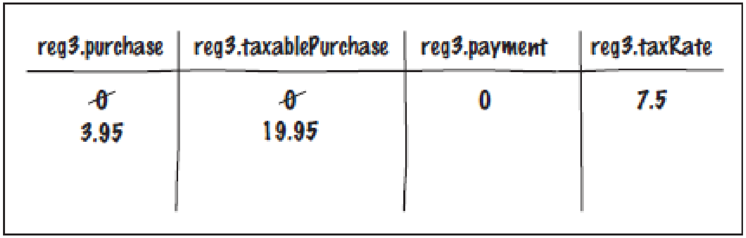
Problem Solving: Tracing Objects

- Example: `CashRegister` class enhancement

The code:

```
CashRegister reg3(7.5); // 7.5 percent sales tax
reg3.recordPurchase(3.95); // Not taxable
reg3.recordTaxablePurchase(19.95); // Taxable
```

The card:



reg3.purchase	reg3.taxablePurchase	reg3.payment	reg3.taxRate
~~0~~ 3.95	~~0~~ 19.95	0	7.5

Local Variables

- **Local variables** are declared in the body of a method:

```
public double giveChange()
{
   double change = payment - purchase;
   purchase = 0;
   payment = 0;
   return change;
}
```

- When a method exits, its local variables are removed.
- **Parameter variables** are declared in the header of a method:

```
public void enterPayment(double amount)
```

Local Variables

- Local and parameter variables belong to methods:

  When a method runs, its local and parameter variables come to life When the method exits, they are removed immediately

- Instance variables belong to objects, not methods:

  When an object is constructed, its instance variables are created

  The instance variables stay alive until no method uses the object any longer

- Instance variables are initialized to a default value:

  Numbers are initialized to 0

  Object references are set to a special value called `null`

  - A `null` object reference refers to no object at all

- You must initialize local variables:

  The compiler complains if you do not

The this Reference

- Two types of inputs are passed when a method is called:

  The object on which you invoke the method

  The method arguments

- In the call `momsSavings.deposit(500)` the method needs to know:

  The account object (`momsSavings`)

  The amount being deposited (`500`)

- The **implicit parameter** of a method is the object on which the method is invoked.
- All other parameter variables are called **explicit parameters**.

The this Reference

- Look at this method:

```
public void deposit(double amount)
{
 balance = balance + amount;
}
```

amount is the explicit parameter

The implicit parameter(momSavings) is not seen

balance means momSavings.balance

- When you refer to an instance variable inside a method, it means the instance variable of the implicit parameter.

The `this` Reference

- The `this` reference denotes the implicit parameter

```
balance = balance + amount;
```

- actually means

```
this.balance = this.balance + amount;
```

- When you refer to an instance variable in a method, the compiler automatically applies it to the `this` reference.

The this Reference

- Some programmers feel that inserting the `this` reference before every instance variable reference makes the code clearer:

```
public BankAccount(double initialBalance)
{
   this.balance = initialBalance;
}
```

The this Reference

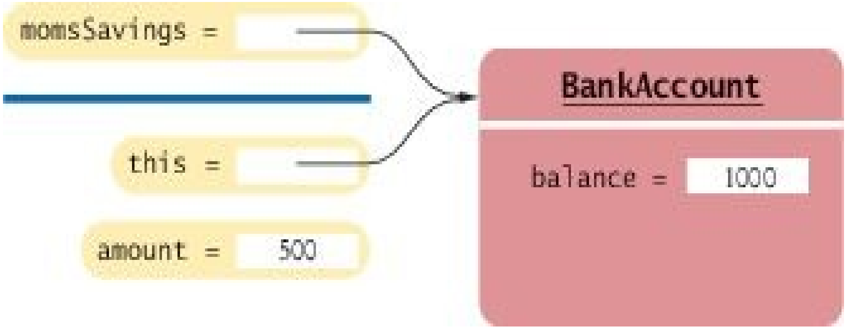


Figure 7 The Implicit Parameter of a Method Call

The this Reference

- The this reference can be used to distinguish between instance variables and local or parameter variables:

```
public BankAccount(double balance)
{
   this.balance = balance;
}
```

- A local variable shadows an instance variable with the same name.

You can access the instance variable name through the this reference.

- In Java, local and parameter variables are considered first when looking up variable names.
- Statement

```
this.balance = balance;
```

means: "Set the instance variable balance to the parameter variable balance".

The this Reference

- A method call without an implicit parameter is applied to the same object. Example:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      withdraw(10); // Withdraw $10 from this account
   }
}
```

- The implicit parameter of the `withdraw` method is the (invisible) implicit parameter of the `monthlyFee` method
- You can use the `this` reference to make the method easier to read:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      this.withdraw(10); // Withdraw $10 from this account
   }
}
```

Drawing Cars

Goal: draw two cars - one in top-left corner of window, and another in the bottom right.

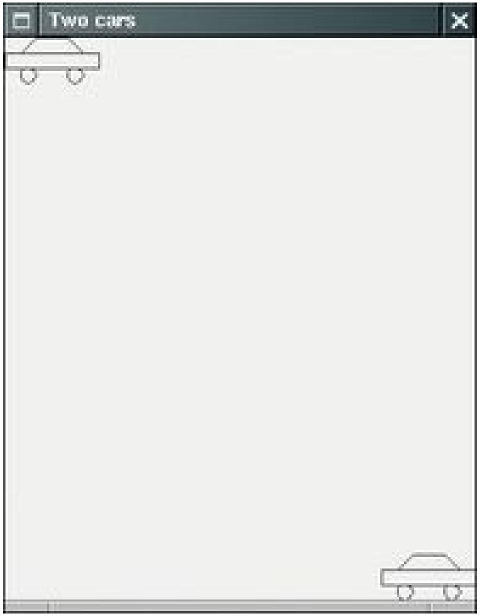


Figure 8 The Car Component Draws Two Car Shapes

Plan Complex Shapes on Graph Paper

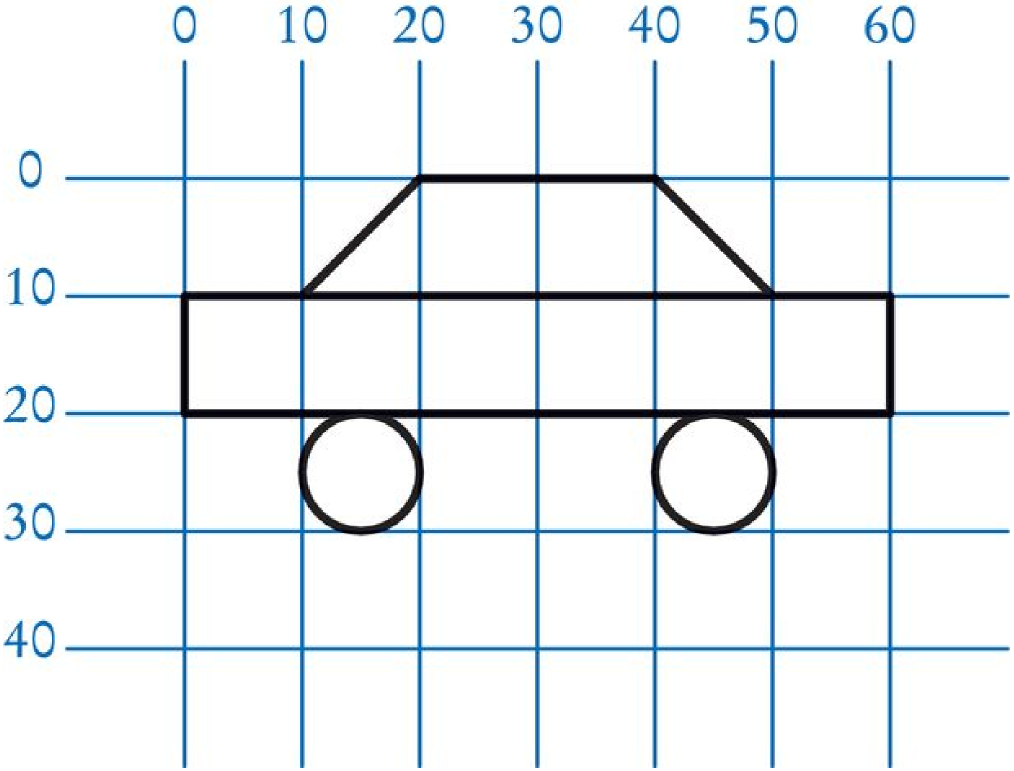


Figure 9 Using Graph Paper to Find Shape Coordinates

Drawing Cars

The program that produces the drawing is composed of three classes:

- The `Car` class is responsible for drawing a single car.
    - Two objects of this class are constructed, one for each car.
- The `CarComponent` class displays the drawing.
- The `CarViewer` class shows a frame that contains a `CarComponent`.

Drawing Cars

- The `paintComponent` method of the `CarComponent` class draws the two cars.
- To compute bottom right position:

```
Car car1 = new Car(0, 0);
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```

  - `getWidth` and `getHeight` return the dimensions of the CarComponent
  - Subtract the dimensions of the car to determine the position of car2:
- When window is resized
  - `paintComponent` is called
  - car position is recomputed using current dimensions

section_8/Car.java

```
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;

/**
   A car shape that can be positioned anywhere on the screen.
*/
public class Car
{
   private int xLeft;
   private int yTop;

   /**
      Constructs a car with a given top left corner.
      @param x the x coordinate of the top left corner
      @param y the y coordinate of the top left corner
   */
   public Car(int x, int y)
   {
      xLeft = x;
      yTop = y;
   }

   /**
      Draws the car.
      @param g2 the graphics context
   */
   public void draw(Graphics2D g2)
   {
     Rectangle body = new Rectangle(xLeft, yTop + 10, 60, 10);
      Ellipse2D.Double frontTire
        = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
      Ellipse2D.Double rearTire
```

section_8/CarComponent.java

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JComponent;

/**
This component draws two car shapes.
*/
public class CarComponent extends JComponent
{
public void paintComponent(Graphics g)
{
Graphics2D g2 = (Graphics2D) g;

Car car1 = new Car(0, 0);

int x = getWidth() - 60;
int y = getHeight() - 30;

Car car2 = new Car(x, y);

car1.draw(g2);
car2.draw(g2);
}
}
```

section_8/CarViewer.java

```java
import javax.swing.JFrame;

public class CarViewer
{
public static void main(String[] args)
{
JFrame frame = new JFrame();

frame.setSize(300, 400);
frame.setTitle("Two cars");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

CarComponent component = new CarComponent();
frame.add(component);

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

Revisiting System.out.println()

- Let’s review this code:

```
import java.lang.Math;
public class MyPoint {
            int x, y;
            public static MyPoint origin = new MyPoint(); //NOTE
            public MyPoint(int givenX, int givenY) {
                     this.x = givenX;
                     this.y = givenY;
            }
            public MyPoint() {}
            public double distanceTo(MyPoint other) {
                     return Math.sqrt( Math.pow( x - other.x, 2 ) +
                                        Math.pow( y - other.y, 2 ) );
            }
}
```

Program Run:

```
Welcome to DrJava.
> MyPoint a = new MyPoint(1,1)
> MyPoint.origin.distanceTo(a)
1.4142135623730951
```

- Here is how this works:
  - My class contains a static instance (or object) of itself whose method I am using when I call `distanceTo` method.
  - This is exactly what System.out.println() does!