

ICOM 4015: Advanced Programming

Lecture 3

Chapter Three: Implementing Classes

CAY HORSTMANN



Chapter Three: Implementing Classes

Big Java by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.

Chapter Goals

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

Black Boxes

- A black box magically does its thing
- Hides its inner workings
- Encapsulation: the hiding of unimportant details
- What is the right *concept* for each particular black box?
- Concepts are discovered through abstraction
- Abstraction: taking away inessential features, until only the essence of the concept remains
- In *object-oriented programming* the black boxes from which a program is manufactured are called objects

Levels of Abstraction: A Real Life Example

- Black boxes in a car: transmission, electronic control module, etc.

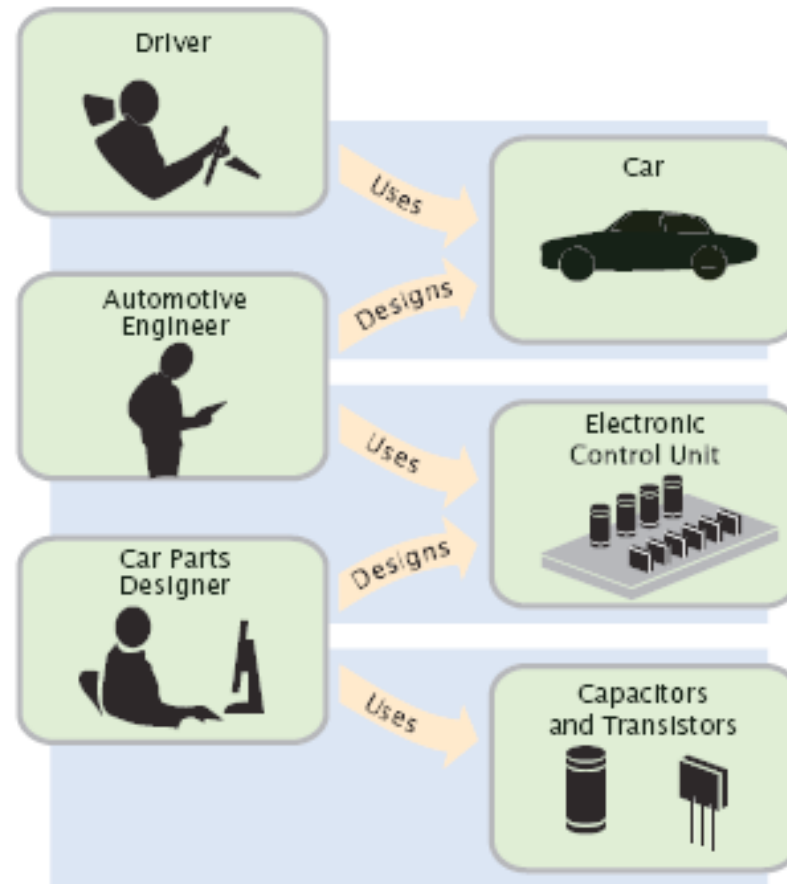


Figure 1

Levels of Abstraction in Automotive Design ava by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.

Levels of Abstraction: A Real Life Example

- Users of a car do not need to understand how black boxes work
- Interaction of a black box with outside world is well-defined
 - *Drivers interact with car using pedals, buttons, etc.*
 - *Mechanic can test that engine control module sends the right firing signals to the spark plugs*
 - *For engine control module manufacturers, transistors and capacitors are black boxes magically produced by an electronics component manufacturer*
- Encapsulation leads to efficiency:
 - *Mechanic deals only with car components (e.g. electronic control module), not with sensors and transistors*
 - *Driver worries only about interaction with car (e.g. putting gas in the tank), not about motor or electronic control module*

Levels of Abstraction: Software Design

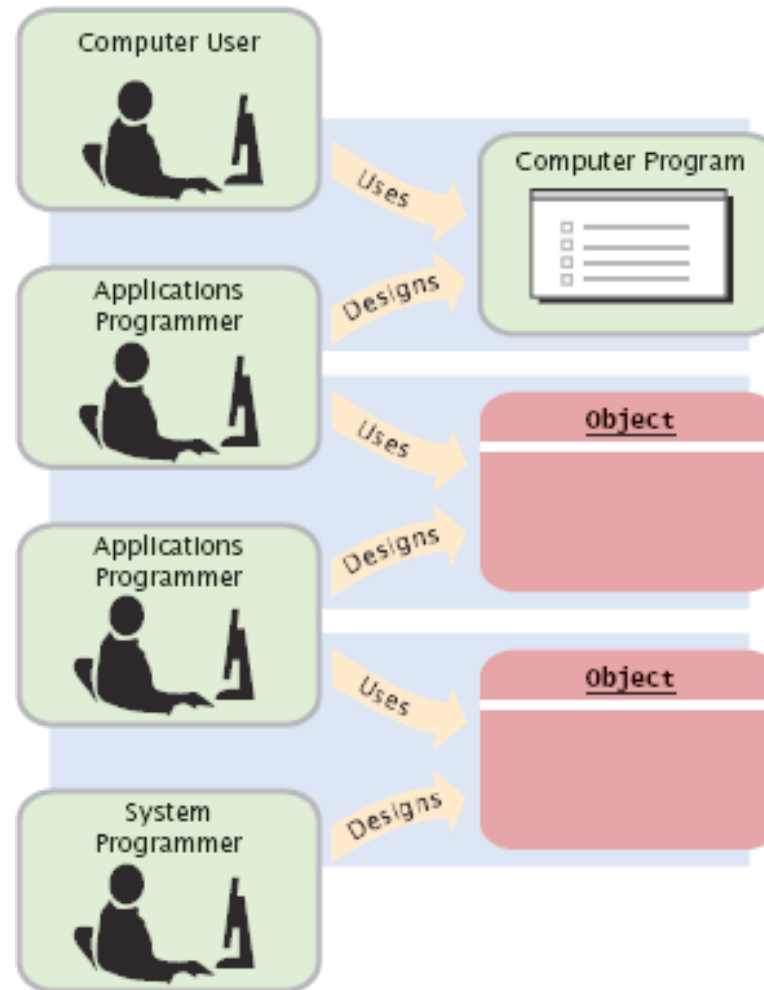


Figure 2

Levels of Abstraction in Software Design

Levels of abstraction: Software Design

- Old times: computer programs manipulated primitive types such as numbers and characters
- Manipulating too many of these primitive quantities is too much for programmers and leads to errors
- Solution: Encapsulate routine computations to software black boxes
- Abstraction used to invent higher-level data types
- In object-oriented programming, objects are black boxes
- Encapsulation: Programmer using an object knows about its behavior, but not about its internal structure

Continued

Levels of abstraction: Software Design (cont.)

- In software design, you can design good and bad abstractions with equal facility; understanding what makes good design is an important part of the education of a software engineer
- First, define behavior of a class; then, implement it

Self Check 3.1

In Chapters 1 and 2, you used `System.out` as a black box to cause output to appear on the screen. Who designed and implemented `System.out`?

Answer: The programmers who designed and implemented the Java library.

Self Check 3.2

Suppose you are working in a company that produces personal finance software. You are asked to design and implement a class for representing bank accounts. Who will be the users of your class?

Answer: Other programmers who work on the personal finance application.

Specifying the Public Interface of a Class

Behavior of bank account (abstraction):

- deposit money
- withdraw money
- get balance

Specifying the Public Interface of a Class: Methods

Methods of `BankAccount` class:

- `deposit`
- `withdraw`
- `getBalance`

We want to support method calls such as the following:

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

Specifying the Public Interface of a Class: Method Definition

access specifier (such as `public`)

- return type (such as `String` or `void`)
- method name (such as `deposit`)
- list of parameters (`double amount` for `deposit`)
- method body in `{ }`

Examples:

- `public void deposit(double amount) { . . . }`
- `public void withdraw(double amount) { . . . }`
- `public double getBalance() { . . . }`

Syntax 3.1 Method Definition

```
accessSpecifier returnType methodName(parameterType  
parameterName, . . .)  
{  
    method body  
}
```

Example:

```
public void deposit(double amount)  
{  
    . . .  
}
```

Purpose:

To define the behavior of a method.

Specifying the Public Interface of a Class: Constructor Definition

- A constructor initializes the instance fields
- Constructor name = class name

```
public BankAccount()  
{  
    // body--filled in later  
}
```

- Constructor body is executed when new object is created
- Statements in constructor body will set the internal data of the object that is being constructed
- All constructors of a class have the same name
- Compiler can tell constructors apart because they take different parameters

Syntax 3.2 Constructor Definition

```
accessSpecifier ClassName(parameterType parameterName, . . .)
{
    constructor body
}
```

Example:

```
public BankAccount(double initialBalance)
{
    . . .
}
```

Purpose:

To define the behavior of a constructor.

BankAccount **Public Interface**

The public constructors and methods of a class form the *public interface* of the class.

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

Continued

Big Java by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

BankAccount Public Interface (cont.)

```
// 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
}
// private fields--filled in later
}
```

Syntax 3.3 Class Definition

```
accessSpecifier class ClassName
{
    constructors
    methods
    fields
}
```

Example:

```
public class BankAccount
{
    public BankAccount(double initialBalance) {. . .}
    public void deposit(double amount) {. . .}
    . . .
}
```

Purpose:

To define a class, its public interface, and its implementation details.

Self Check 3.3

How can you use the methods of the public interface to *empty* the `harrysChecking` bank account?

Answer:

```
harrysChecking.withdraw(harrysChecking.getBalance())
```

Self Check 3.4

Suppose you want a more powerful bank account abstraction that keeps track of an *account number* in addition to the balance. How would you change the public interface to accommodate this enhancement?

Answer: Add an `accountNumber` parameter to the constructors, and add a `getAccountNumber` method. There is no need for a `setAccountNumber` method – the account number never changes after construction.

Commenting the Public Interface

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

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

Class Comment

```
/**  
    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*
 - *every return value.*

Javadoc Method Summary

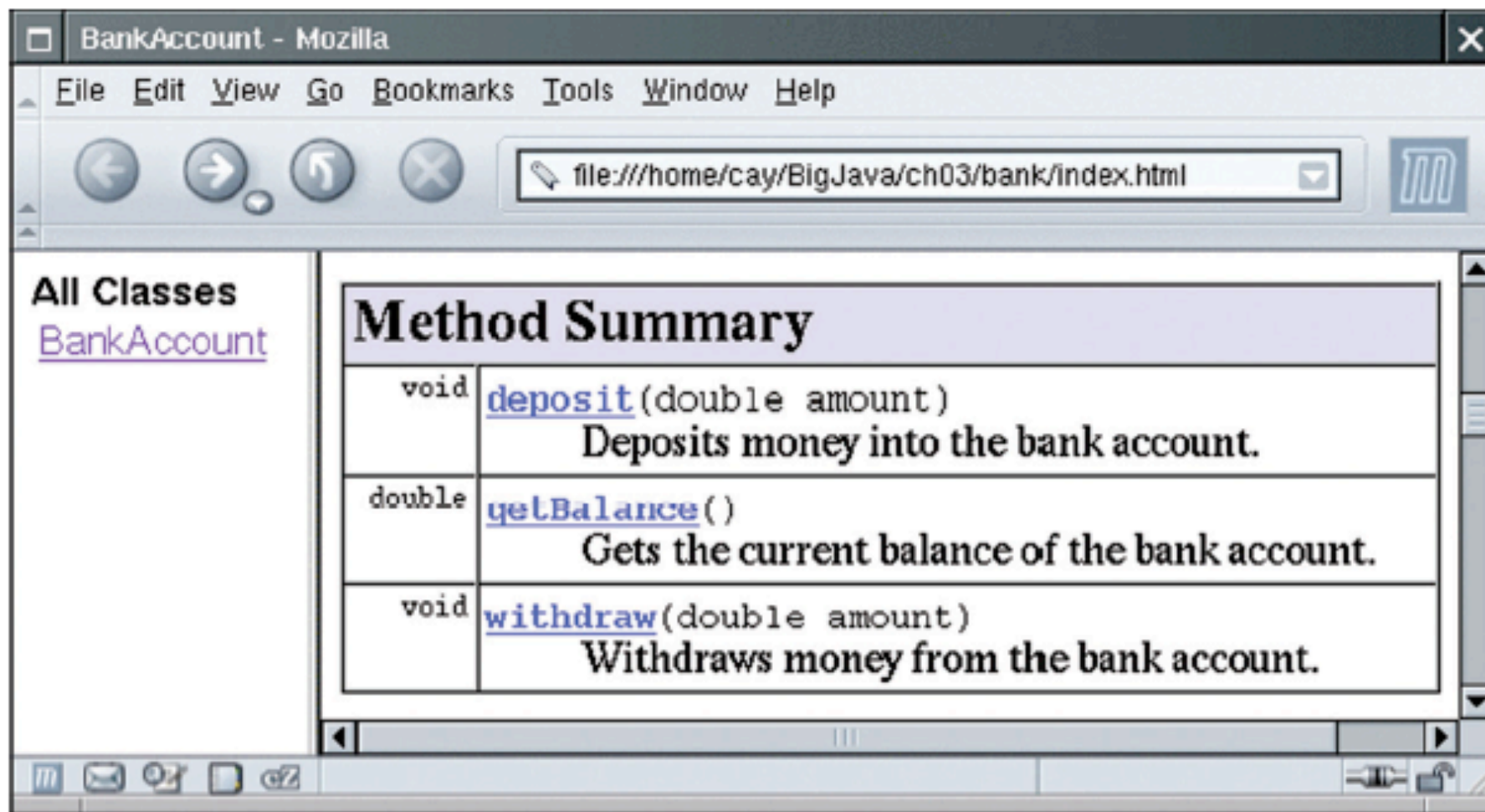


Figure 3 A Method Summary Generated by javadoc

Javadoc Method Detail

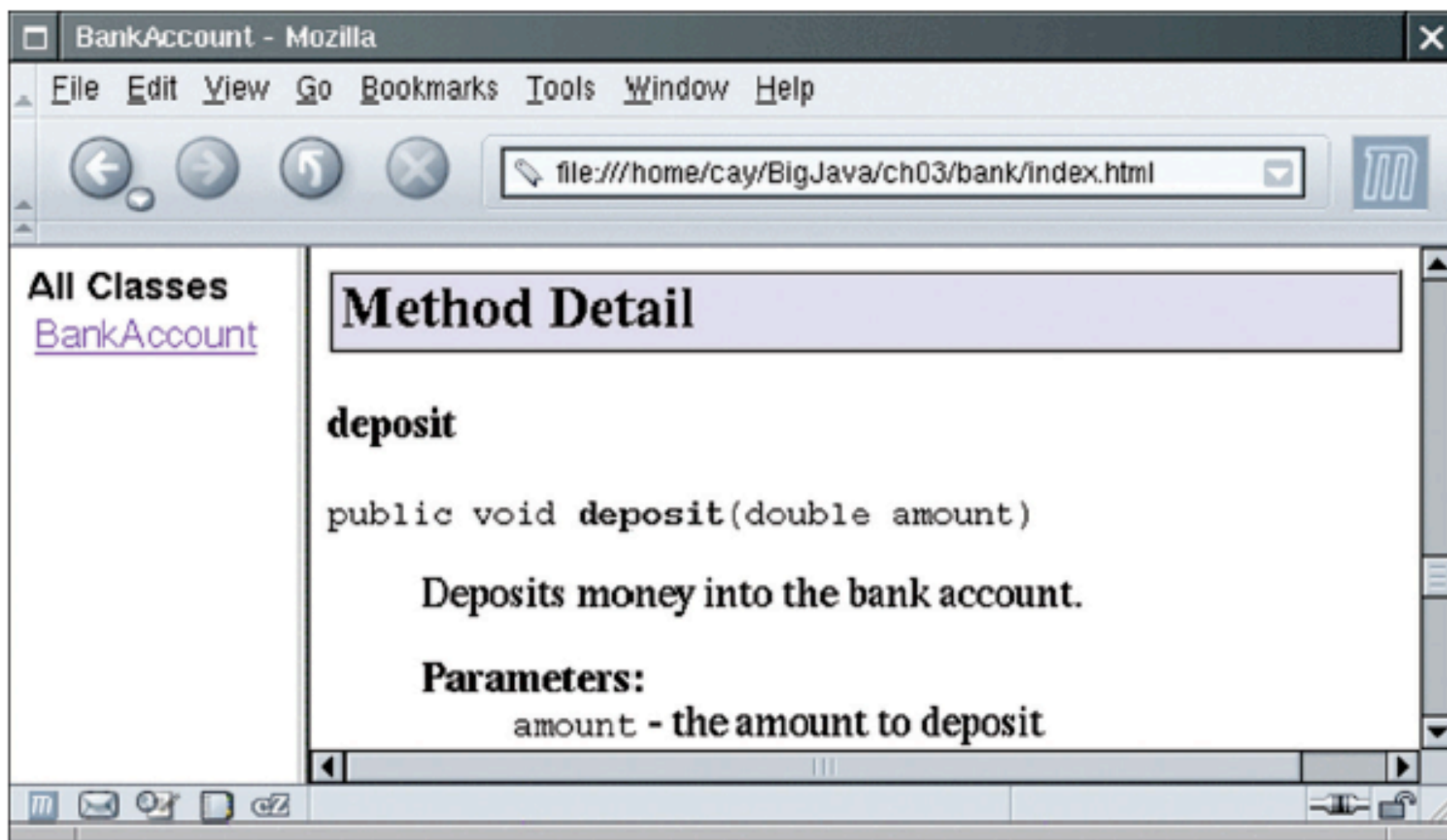


Figure 4 Method Detail Generated by javadoc

Self Check 3.5

Suppose we enhance the `BankAccount` class so that each account has an account number. Supply a documentation comment for the constructor

```
public BankAccount(int accountNumber, double
    initialBalance)
```

Answer:

```
/**
 * Constructs a new bank account with a given initial
 * balance.
 * @param accountNumber the account number for
 * this account
 * @param initialBalance the initial balance
 * for this account
 */
```

Self Check 3.6

Why is the following documentation comment questionable?

```
/**
 * Each account has an account number.
 * @return the account number of this account
 */
public int getAccountNumber()
```

Answer: The first sentence of the method description should describe the method – it is displayed in isolation in the summary table.

Instance Fields

- An object stores its data in instance fields
- Field: a technical term for a storage location inside a block of memory
- Instance of a class: an object of the class
- The class declaration specifies the instance fields

```
public class BankAccount
{
    . . .
    private double balance;
}
```

Instance Fields

- An instance field declaration consists of the following parts:
 - *access specifier (usually `private`)*
 - *type of variable (such as `double`)*
 - *name of variable (such as `balance`)*
- Each object of a class has its own set of instance fields
- You should declare all instance fields as private

Instance Fields

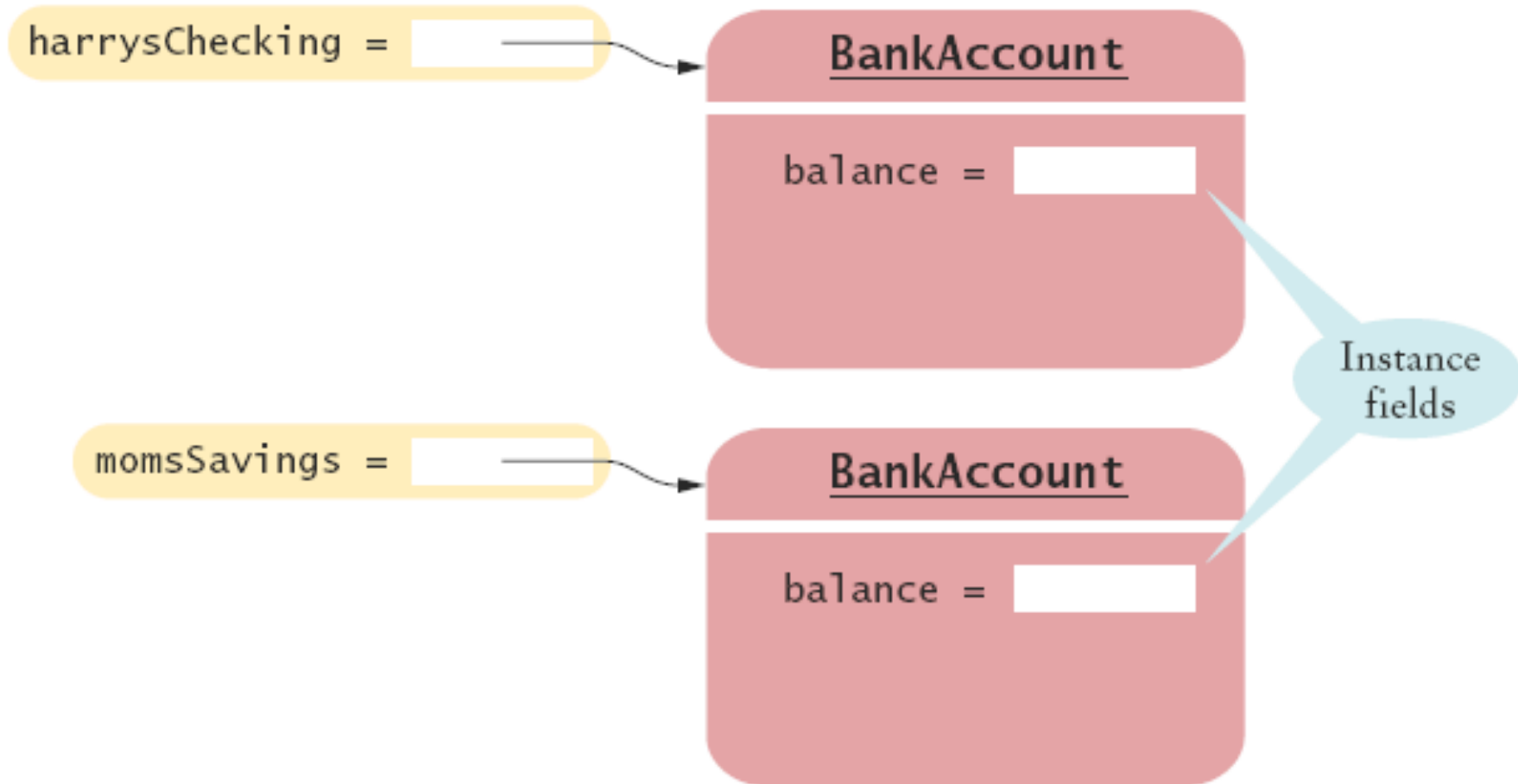


Figure 5 Instance Fields

Syntax 3.4 Instance Field Declaration

```
accessSpecifier class ClassName
{
    . . .
    accessSpecifier fieldType fieldName;
    . . .
}
```

Example:

```
public class BankAccount
{
    . . .
    private double balance;
    . . .
}
```

Purpose:

To define a field that is present in every object of a class.

Accessing Instance Fields

- The `deposit` method of the `BankAccount` class can access the private instance field:

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

Continued

Big Java by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

Accessing Instance Fields (cont.)

- Other methods cannot:

```
public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momsSavings = new BankAccount(1000);
        . . .
        momsSavings.balance = -1000; // ERROR
    }
}
```

- *Encapsulation* is the process of hiding object data and providing methods for data access
- To encapsulate data, declare instance fields as `private` and define public methods that access the fields

Self Check 3.7

Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance fields?

Answer: An instance field

```
private int accountNumber;
```

needs to be added to the class.

Self Check 3.8

What are the instance fields of the `Rectangle` class?

Answer: There are four fields, `x`, `y`, `width` and `height`. All four fields have type `int`.

Implementing Constructors

- Constructors contain instructions to initialize the instance fields of an object

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

Constructor Call Example

- `BankAccount harrysChecking = new BankAccount(1000);`
 - *Create a new object of type `BankAccount`*
 - *Call the second constructor (since a construction parameter is supplied)*
 - *Set the parameter variable `initialBalance` to 1000*
 - *Set the `balance` instance field of the newly created object to `initialBalance`*
 - *Return an object reference, that is, the memory location of the object, as the value of the `new` expression*
 - *Store that object reference in the `harrysChecking` variable*

Implementing Methods

- Some methods do not return a value

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

- Some methods return an output value

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

Method Call Example

- `harrysChecking.deposit(500);`
 - *Set the parameter variable `amount` to 500*
 - *Fetch the `balance` field of the object whose location is stored in `harrysChecking`*
 - *Add the value of `amount` to `balance` and store the result in the variable `newBalance`*
 - *Store the value of `newBalance` in the `balance` instance field, overwriting the old value*

Syntax 3.5 The `return` Statement

```
return expression;  
or  
return;
```

Example:

```
return balance;
```

Purpose:

To specify the value that a method returns, and exit the method immediately. The return value becomes the value of the method call expression.

ch03/account/BankAccount.java

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance.
09:     */
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
14:
15:     /**
16:         Constructs a bank account with a given balance.
17:         @param initialBalance the initial balance
18:     */
19:     public BankAccount(double initialBalance)
20:     {
21:         balance = initialBalance;
22:     }
23:
```

Continued

Big Java by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

ch03/account/BankAccount.java (cont.)

```
24:    /**
25:        Deposits money into the bank account.
26:        @param amount the amount to deposit
27:    */
28:    public void deposit(double amount)
29:    {
30:        double newBalance = balance + amount;
31:        balance = newBalance;
32:    }
33:
34:    /**
35:        Withdraws money from the bank account.
36:        @param amount the amount to withdraw
37:    */
38:    public void withdraw(double amount)
39:    {
40:        double newBalance = balance - amount;
41:        balance = newBalance;
42:    }
43:
44:    /**
45:        Gets the current balance of the bank account.
46:        @return the current balance
47:    */
```

Continued
Big Java by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

ch03/account/BankAccount.java (cont.)

```
48:     public double getBalance()  
49:     {  
50:         return balance;  
51:     }  
52:  
53:     private double balance;  
54: }
```

Self Check 3.9

The `Rectangle` class has four instance fields: `x`, `y`, `width`, and `height`. Give a possible implementation of the `getWidth` method.

Answer:

```
public int getWidth()  
{  
    return width;  
}
```

Self Check 3.10

Give a possible implementation of the `translate` method of the `Rectangle` class.

Answer: There is more than one correct answer. One possible implementation is as follows:

```
public void translate(int dx, int dy)
{
    int newX = x + dx;
    x = newX;
    int newY = y + dy;
    y = newY;
}
```

Unit Testing

- *Unit test*: verifies that a class works correctly in isolation, outside a complete program.
- To test a class, use an environment for interactive testing, or write a tester class.
- *Test 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*

Continued

Big Java by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

Unit Testing (cont.)

- 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*

ch03/account/BankAccountTester.java

```
01: /**
02:     A class to test the BankAccount class.
03: */
04: public class BankAccountTester
05: {
06:     /**
07:         Tests the methods of the BankAccount class.
08:         @param args not used
09:     */
10:     public static void main(String[] args)
11:     {
12:         BankAccount harrysChecking = new BankAccount();
13:         harrysChecking.deposit(2000);
14:         harrysChecking.withdraw(500);
15:         System.out.println(harrysChecking.getBalance());
16:         System.out.println("Expected: 1500");
17:     }
18: }
```

Output:

1500

Expected: 1500

Testing With BlueJ

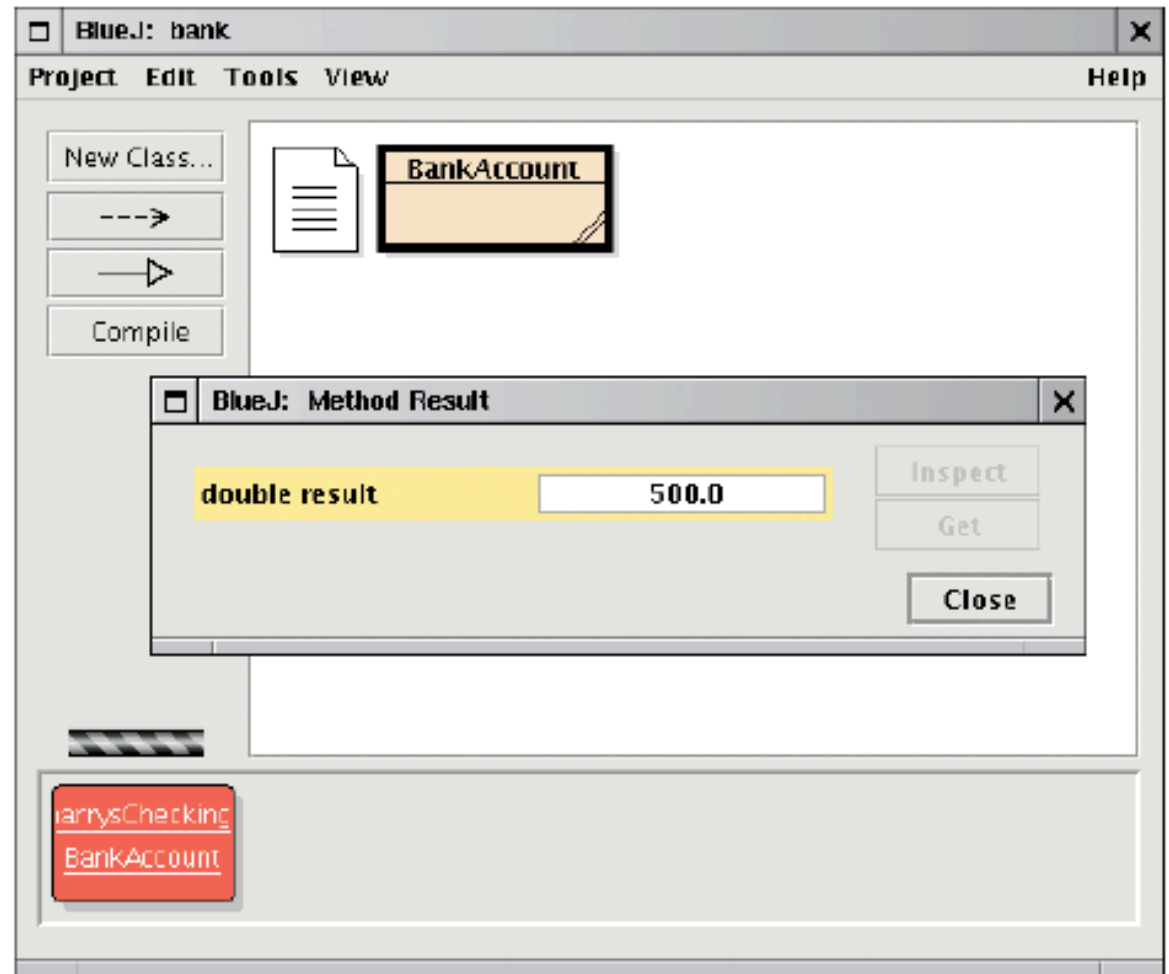


Figure 6

The Return Value of the
getBalance Method in BlueJ

Self Check 3.11

When you run the `BankAccountTester` program, how many objects of class `BankAccount` are constructed? How many objects of type `BankAccountTester`?

Answer: One `BankAccount` object, no `BankAccountTester` object. The purpose of the `BankAccountTester` class is merely to hold the main method.

Self Check 3.12

Why is the `BankAccountTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

Answer: In those environments, you can issue interactive commands to construct `BankAccount` objects, invoke methods, and display their return values.

Categories of Variables

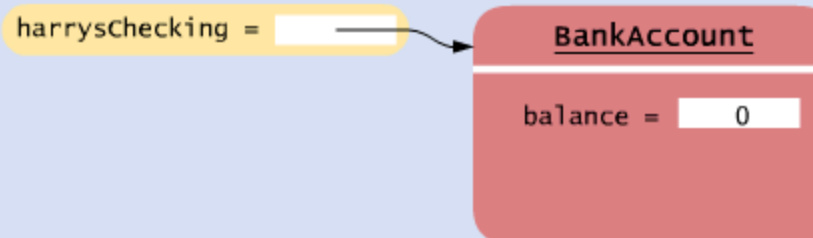
- Categories of variables
 1. *Instance fields* (*balance* in *BankAccount*)
 2. *Local variables* (*newBalance* in *deposit* method)
 3. *Parameter variables* (*amount* in *deposit* method)
- An `instance` field belongs to an object
- The fields stay alive until no method uses the object any longer
- In Java, the *garbage collector* periodically reclaims objects when they are no longer used
- Local and parameter variables belong to a method
- Instance fields are initialized to a default value, but you must initialize local variables

Animation 3.1 –

```
public static void main(String[] args)
{
    . . .
    harrysChecking.deposit(500);
    . . .
}

. . .

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

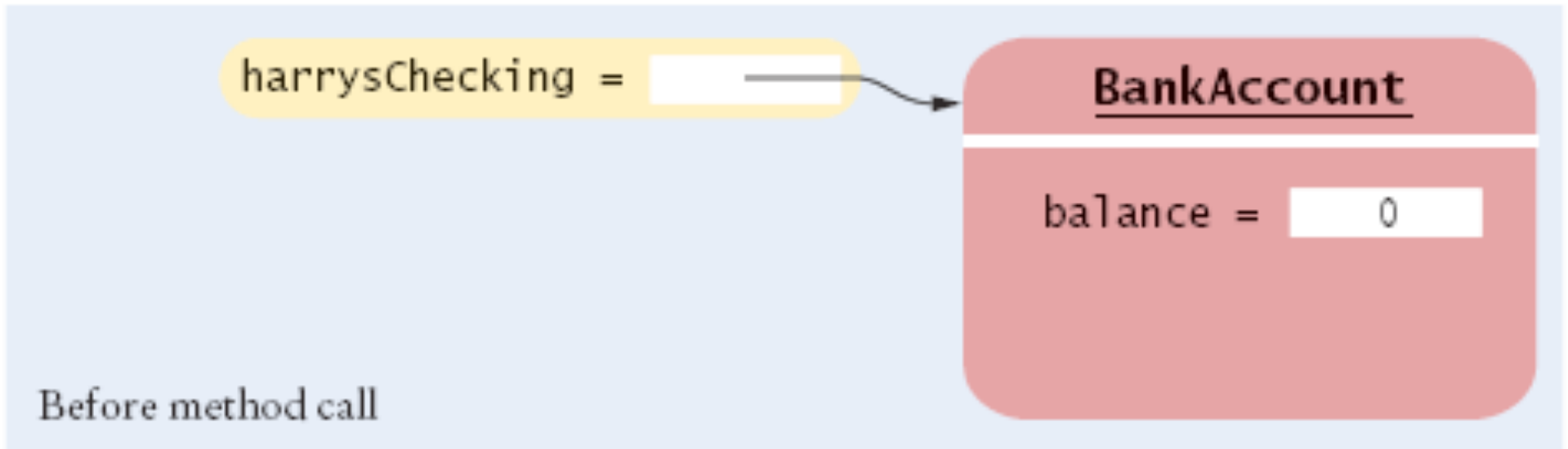


This animation demonstrates the lifetime of local variables and parameter variables.



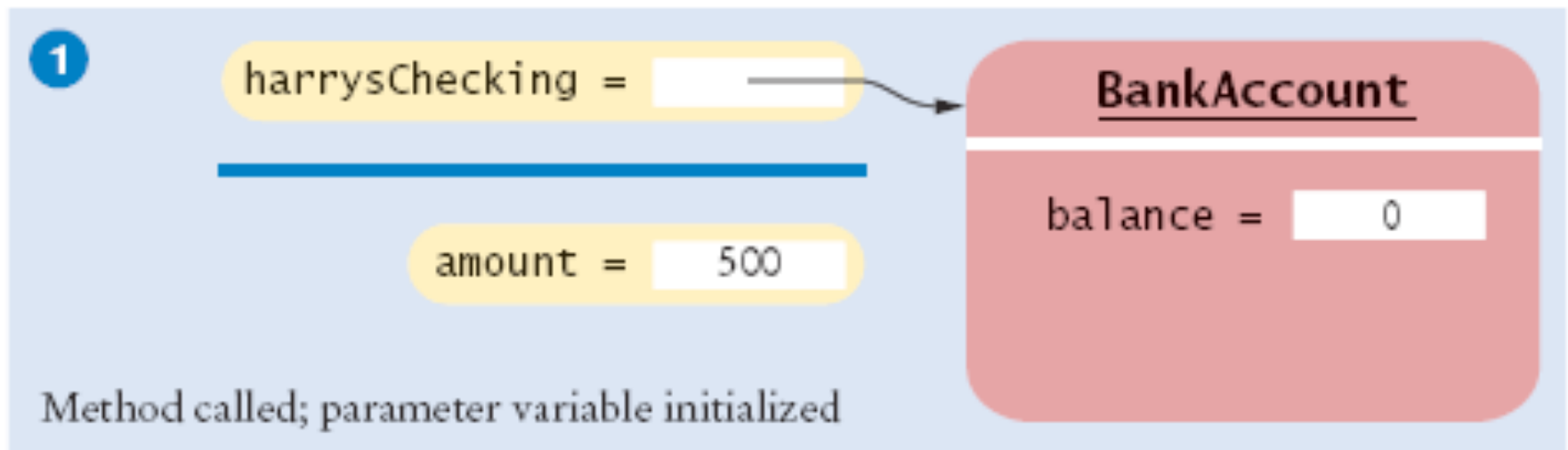
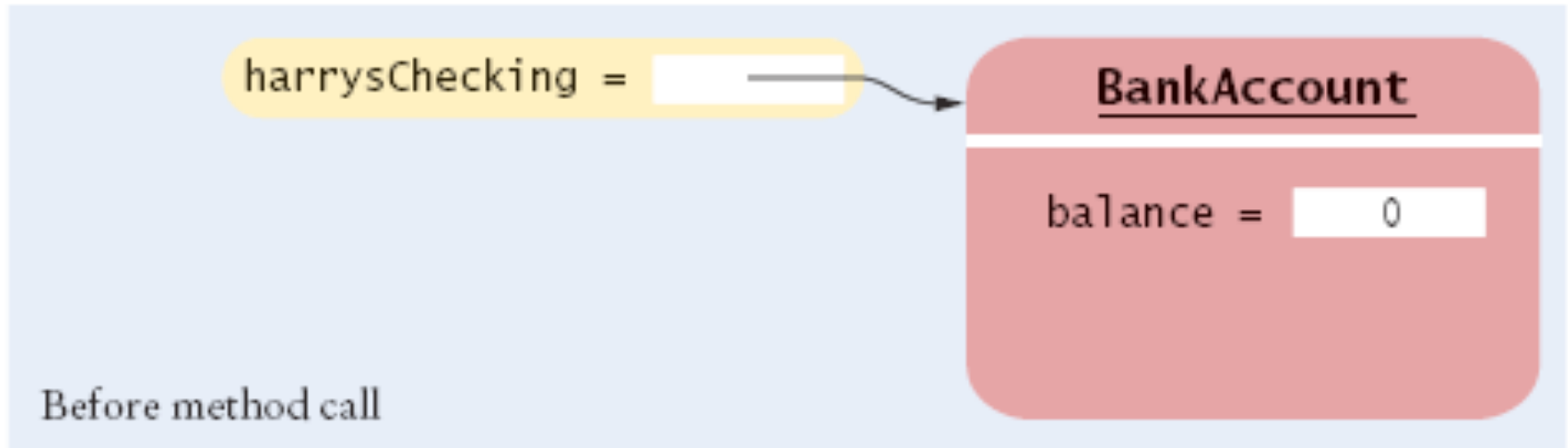
Lifetime of Variables – Calling Method `deposit`

```
harrysChecking.deposit(500);
```



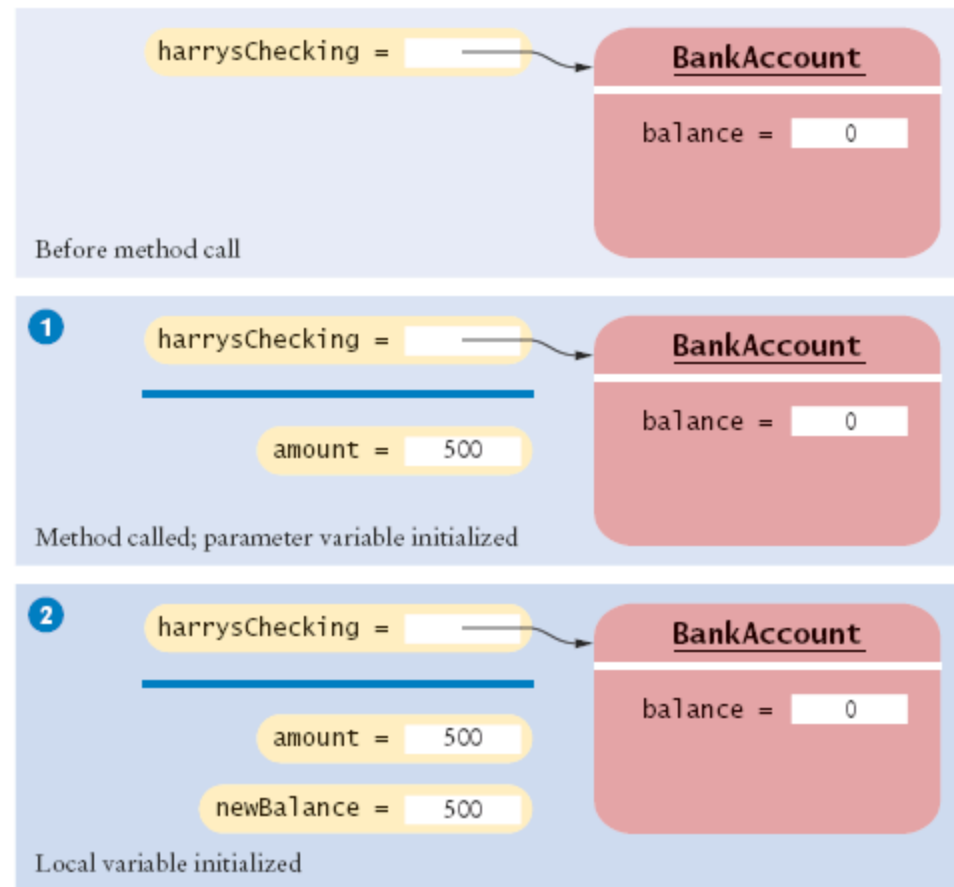
Lifetime of Variables – Calling Method `deposit`

```
harrysChecking.deposit(500); ①
```



Lifetime of Variables – Calling Method `deposit`

```
harrysChecking.deposit(500); 1  
double newBalance = balance + amount; 2
```



Lifetime of Variables – Calling Method `deposit`

```
harrysChecking.deposit(500); ①  
double newBalance = balance + amount; ②  
balance = newBalance; ③
```

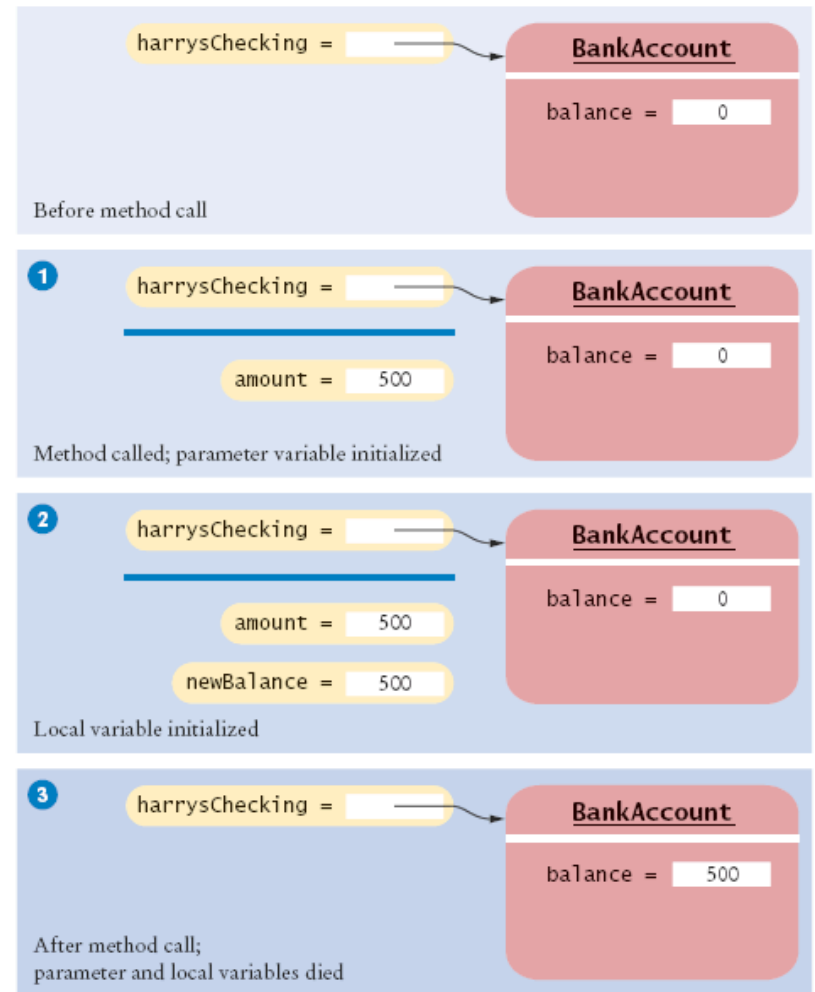


Figure 7 Lifetime of Variables

Self Check 3.13

What do local variables and parameter variables have in common? In which essential aspect do they differ?

Answer: Variables of both categories belong to methods – they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the call values; local variables must be explicitly initialized.

Self Check 3.14

During execution of the `BankAccountTester` program in the preceding section, how many instance fields, local variables, and parameter variables were created, and what were their names?

Answer: One instance field, named `balance`. Three local variables, one named `harrysChecking` and two named `newBalance` (in the `deposit` and `withdraw` methods); two parameter variables, both named `amount` (in the `deposit` and `withdraw` methods).

Implicit and Explicit Method Parameters

- The implicit parameter of a method is the object on which the method is invoked
- The `this` reference denotes the implicit parameter
- Use of an instance field name in a method denotes the instance field of the implicit parameter

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

Continued

Implicit and Explicit Method Parameters (cont.)

- `balance` is the balance of the object to the left of the dot:

```
momsSavings.withdraw(500)
```

means

```
double newBalance = momsSavings.balance - amount;  
>momsSavings.balance = newBalance;
```

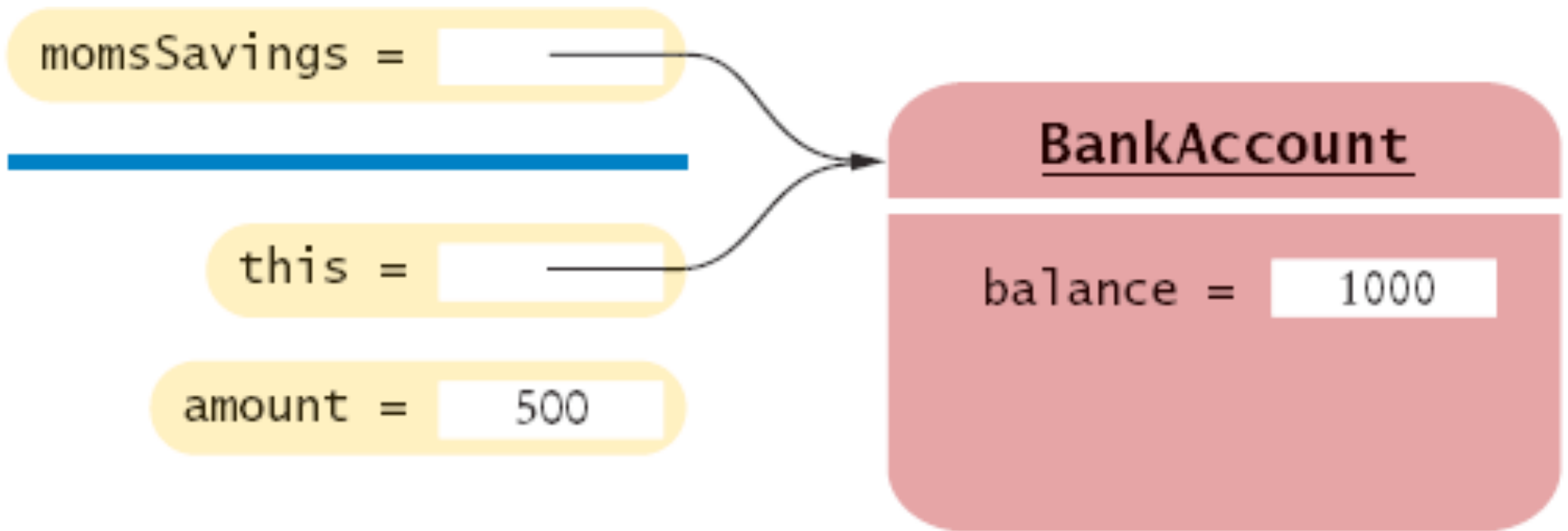
Implicit Parameters and `this`

- Every method has one implicit parameter
- The implicit parameter is always called `this`
- Exception: Static methods do not have an implicit parameter (more on Chapter 8)
- ```
double newBalance = balance + amount;
// actually means
double newBalance = this.balance + amount;
```
- When you refer to an instance field in a method, the compiler automatically applies it to the `this` parameter

```
momsSavings.deposit(500);
```

## Implicit Parameters and `this`

---



**Figure 8** The Implicit Parameter of a Method Call



## Self Check 3.15

---

How many implicit and explicit parameters does the `withdraw` method of the `BankAccount` class have, and what are their names and types?

**Answer:** One implicit parameter, called `this`, of type `BankAccount`, and one explicit parameter, called `amount`, of type `double`.

## Self Check 3.16

---

In the `deposit` method, what is the meaning of `this.amount`? Or, if the expression has no meaning, why not?

**Answer:** It is not a legal expression. `this` is of type `BankAccount` and the `BankAccount` class has no field named `amount`.

## Self Check 3.17

---

How many implicit and explicit parameters does the `main` method of the `BankAccountTester` class have, and what are they called?

**Answer:** No implicit parameter—the method is static—and one explicit parameter, called `args`.

# Shape Classes

---

Good practice: Make a class for each graphical shape

```
public class Car
{
 public Car(int x, int y)
 {
 // Remember position
 . . .
 }
 public void draw(Graphics2D g2)
 {
 // Drawing instructions
 . . .
 }
}
```

## Drawing Cars

---

- Draw two cars: one in top-left corner of window, and another in the bottom right
- Compute bottom right position, inside `paintComponent` method:

```
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```
- `getWidth` and `getHeight` are applied to object that executes `paintComponent`
- If window is resized `paintComponent` is called and car position recomputed

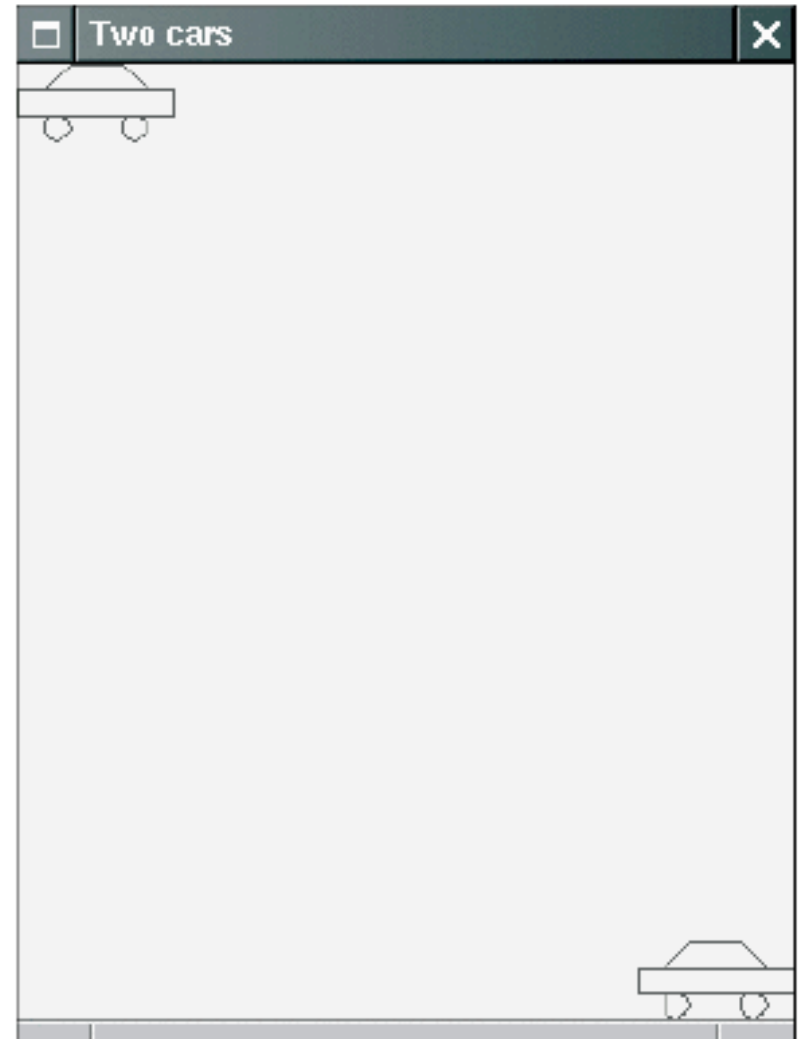
***Continued***

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

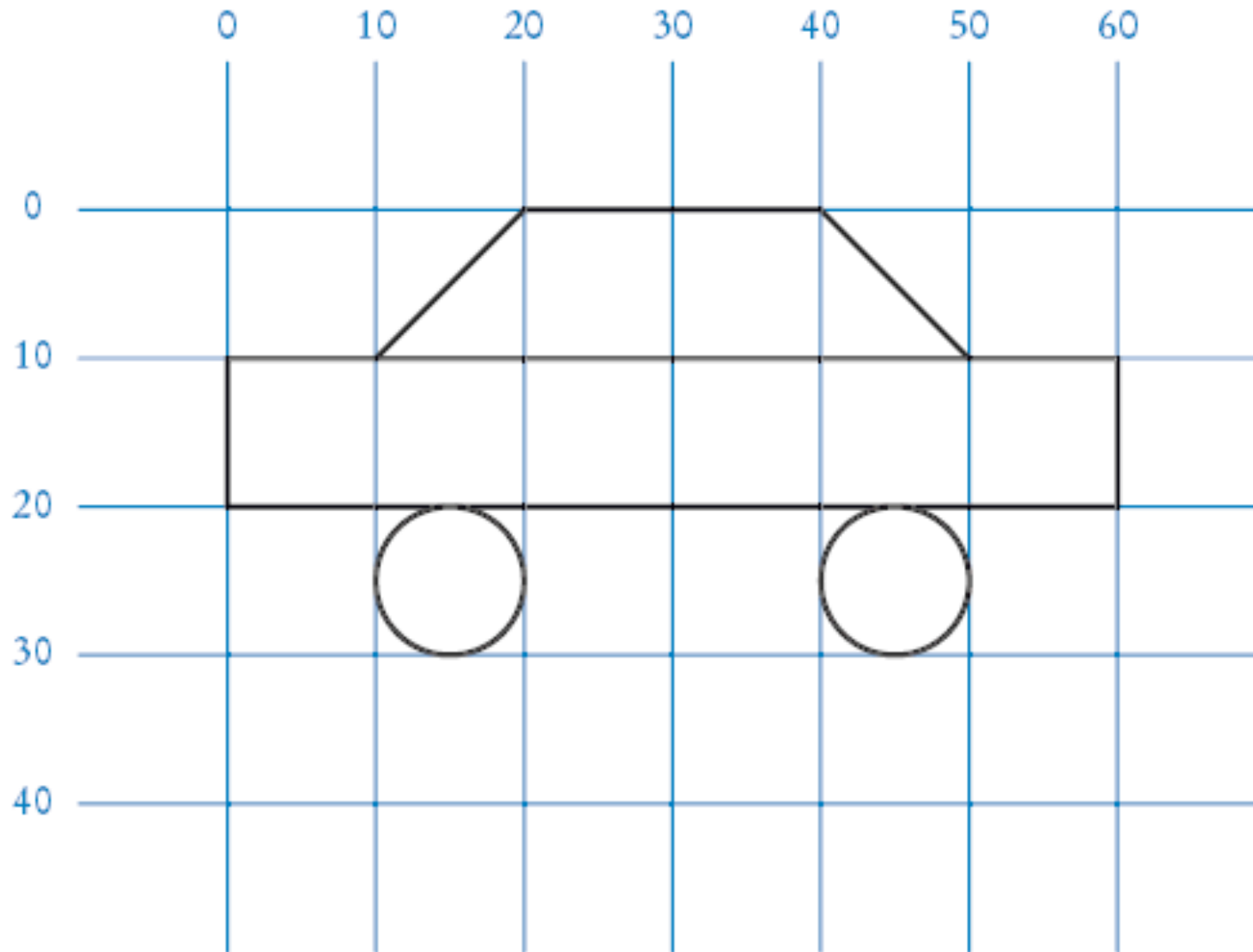
## Drawing Cars (cont.)

---



**Figure 9**  
The Car Component Draws Two Car Shapes

## Plan Complex Shapes on Graph Paper



**Figure 10** Using Graph Paper to Find Shape Coordinates Java by Cay Horstmann  
Copyright © 2000 by John Wiley & Sons. All rights reserved.

## Classes of Car Drawing Program

---

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



## ch03/car/Car.java

---

```
01: import java.awt.Graphics2D;
02: import java.awt.Rectangle;
03: import java.awt.geom.Ellipse2D;
04: import java.awt.geom.Line2D;
05: import java.awt.geom.Point2D;
06:
07: /**
08: A car shape that can be positioned anywhere on the screen.
09: */
10: public class Car
11: {
12: /**
13: Constructs a car with a given top left corner
14: @param x the x coordinate of the top left corner
15: @param y the y coordinate of the top left corner
16: */
17: public Car(int x, int y)
18: {
19: xLeft = x;
20: yTop = y;
21: }
22:
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch03/car/Car.java (cont.)

---

```
23: /**
24: Draws the car.
25: @param g2 the graphics context
26: */
27: public void draw(Graphics2D g2)
28: {
29: Rectangle body
30: = new Rectangle(xLeft, yTop + 10, 60, 10);
31: Ellipse2D.Double frontTire
32: = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
33: Ellipse2D.Double rearTire
34: = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
35:
36: // The bottom of the front windshield
37: Point2D.Double r1
38: = new Point2D.Double(xLeft + 10, yTop + 10);
39: // The front of the roof
40: Point2D.Double r2
41: = new Point2D.Double(xLeft + 20, yTop);
42: // The rear of the roof
43: Point2D.Double r3
44: = new Point2D.Double(xLeft + 40, yTop);
45: // The bottom of the rear windshield
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch03/car/Car.java (cont.)

---

```
46: Point2D.Double r4
47: = new Point2D.Double(xLeft + 50, yTop + 10);
48:
49: Line2D.Double frontWindshield
50: = new Line2D.Double(r1, r2);
51: Line2D.Double roofTop
52: = new Line2D.Double(r2, r3);
53: Line2D.Double rearWindshield
54: = new Line2D.Double(r3, r4);
55:
56: g2.draw(body);
57: g2.draw(frontTire);
58: g2.draw(rearTire);
59: g2.draw(frontWindshield);
60: g2.draw(roofTop);
61: g2.draw(rearWindshield);
62: }
63:
64: private int xLeft;
65: private int yTop;
66: }
```

## ch03/car/CarComponent.java

---

```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import javax.swing.JComponent;
04:
05: /**
06: This component draws two car shapes.
07: */
08: public class CarComponent extends JComponent
09: {
10: public void paintComponent(Graphics g)
11: {
12: Graphics2D g2 = (Graphics2D) g;
13:
14: Car car1 = new Car(0, 0);
15:
16: int x = getWidth() - 60;
17: int y = getHeight() - 30;
18:
19: Car car2 = new Car(x, y);
20:
21: car1.draw(g2);
22: car2.draw(g2);
23: }
24: }
```

## ch03/car/CarViewer.java

---

```
01: import javax.swing.JFrame;
02:
03: public class CarViewer
04: {
05: public static void main(String[] args)
06: {
07: JFrame frame = new JFrame();
08:
09: frame.setSize(300, 400);
10: frame.setTitle("Two cars");
11: frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:
13: CarComponent component = new CarComponent();
14: frame.add(component);
15:
16: frame.setVisible(true);
17: }
18: }
19:
```

## Self Check 3.18

---

Which class needs to be modified to have the two cars positioned next to each other?

**Answer:** `CarComponent`

## Self Check 3.19

---

Which class needs to be modified to have the car tires painted in black, and what modification do you need to make?

**Answer:** In the `draw` method of the `Car` class, call

```
g2.fill(frontTire);
g2.fill(rearTire);
```

## Self Check 3.20

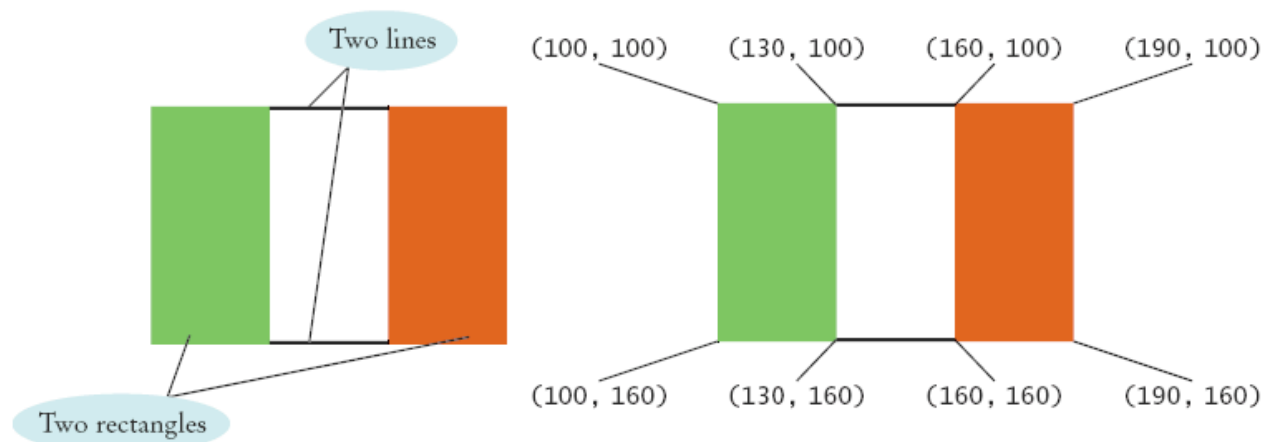
---

How do you make the cars twice as big?

**Answer:** Double all measurements in the `draw` method of the `Car` class.



# Drawing Graphical Shapes



```
Rectangle leftRectangle = new Rectangle(100, 100, 30,
60);
```

```
Rectangle rightRectangle = new Rectangle(160, 100, 30,
60);
```

```
Line2D.Double topLine = new Line2D.Double(130, 100, 160,
100);
```

```
Line2D.Double bottomLine = new Line2D.Double(130, 160,
160, 160);
```