ICOM 4015: Advanced Programming

Lecture 3

Chapter Three: Implementing Classes

Chapter Three: Implementing Classes

Big Java by Cay Horstmann

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

Big Java by Cay Horstmann

Levels of Abstraction: A Real Life Example

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

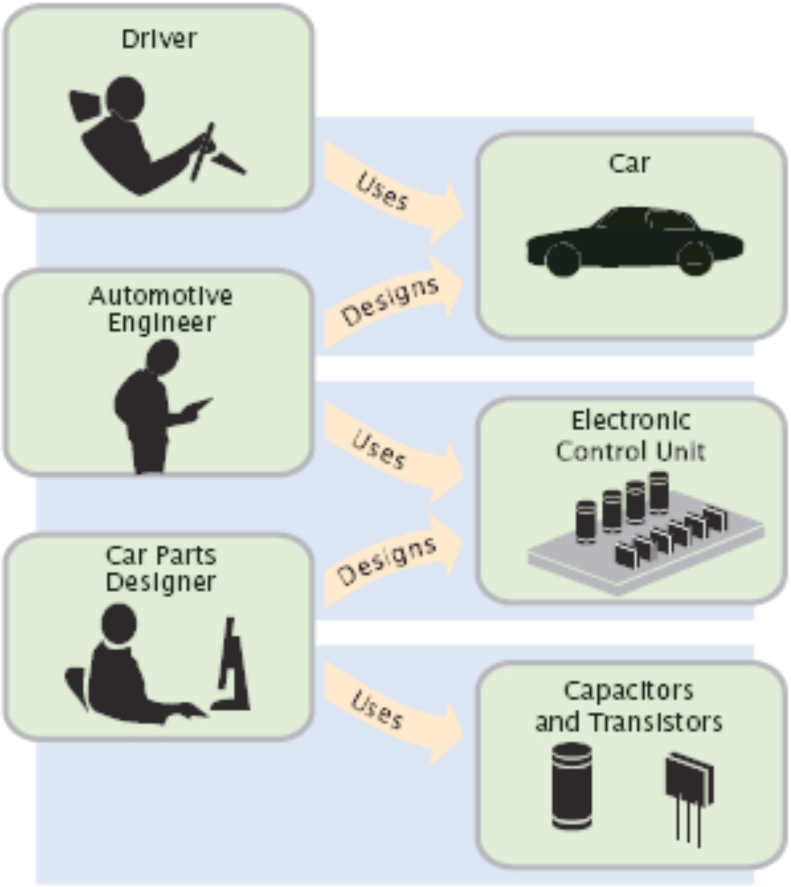


Figure 1

Levels of Abstraction in Automotive Design*ava* by Cay Horstmann

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

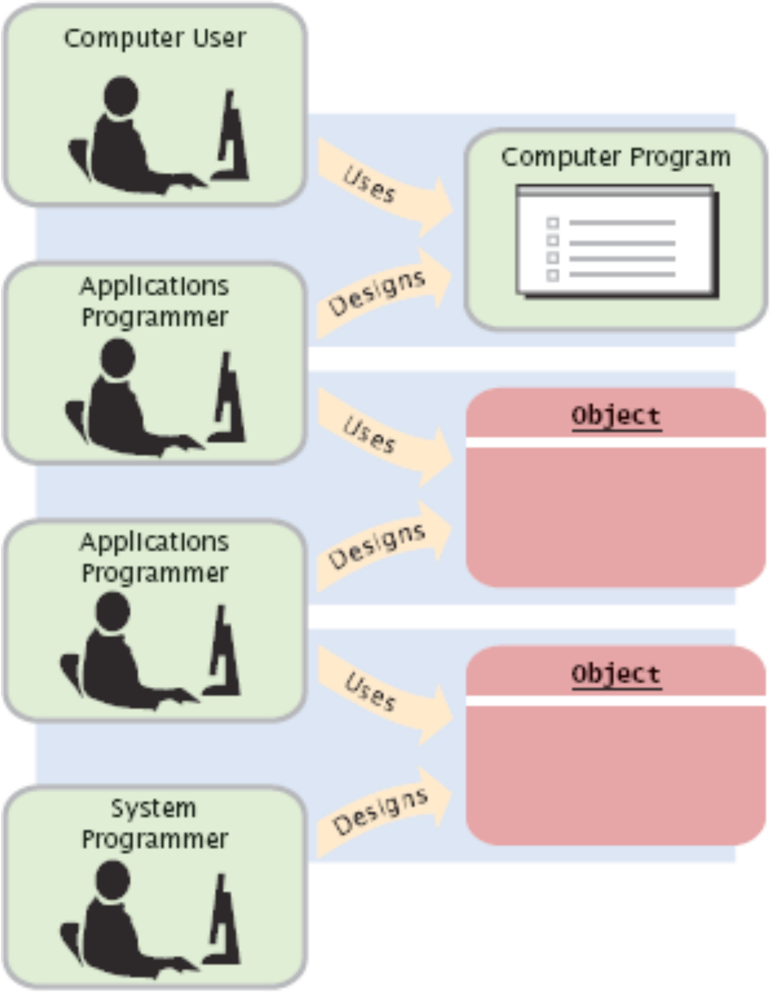


Figure 2

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() { . . . }`

Big Java by Cay Horstmann

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

`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())
```

Big Java by Cay Horstmann

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

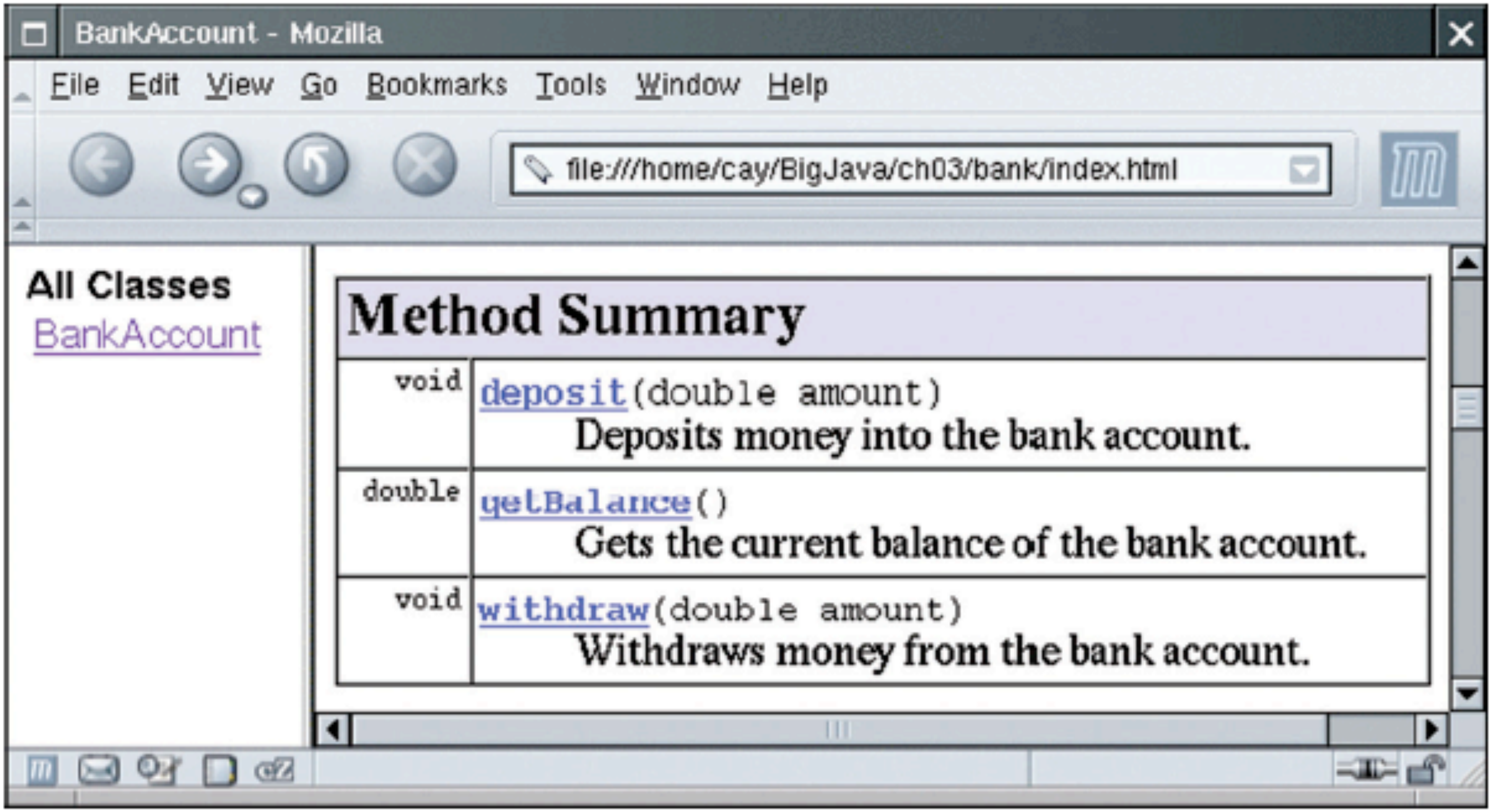
```
/**
   Withdraws money from the bank account.
   @param the amount to withdraw
*/
public void withdraw(double amount)
{
   //implementation filled in later
}
/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   //implementation filled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

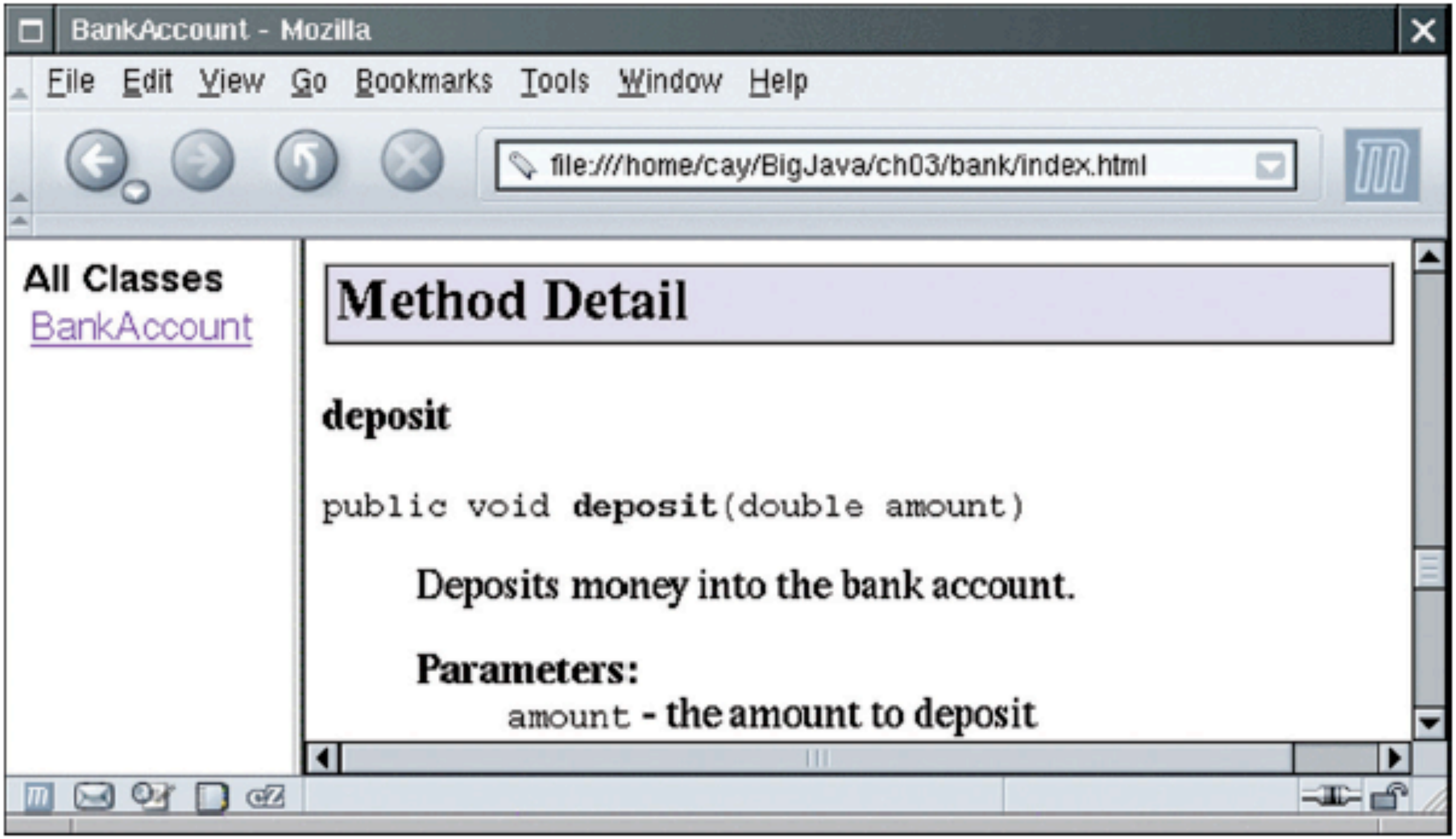

BankAccount - Mozilla
File Edit View Go Bookmarks Tools Window Help
file:///home/cay/BigJava/ch03/bank/index.html
All Classes
BankAccount

Method Summary

`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

Javadoc Method Detail


BankAccount - Mozilla
File Edit View Go Bookmarks Tools Window Help
file:///home/cay/BigJava/ch03/bank/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

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

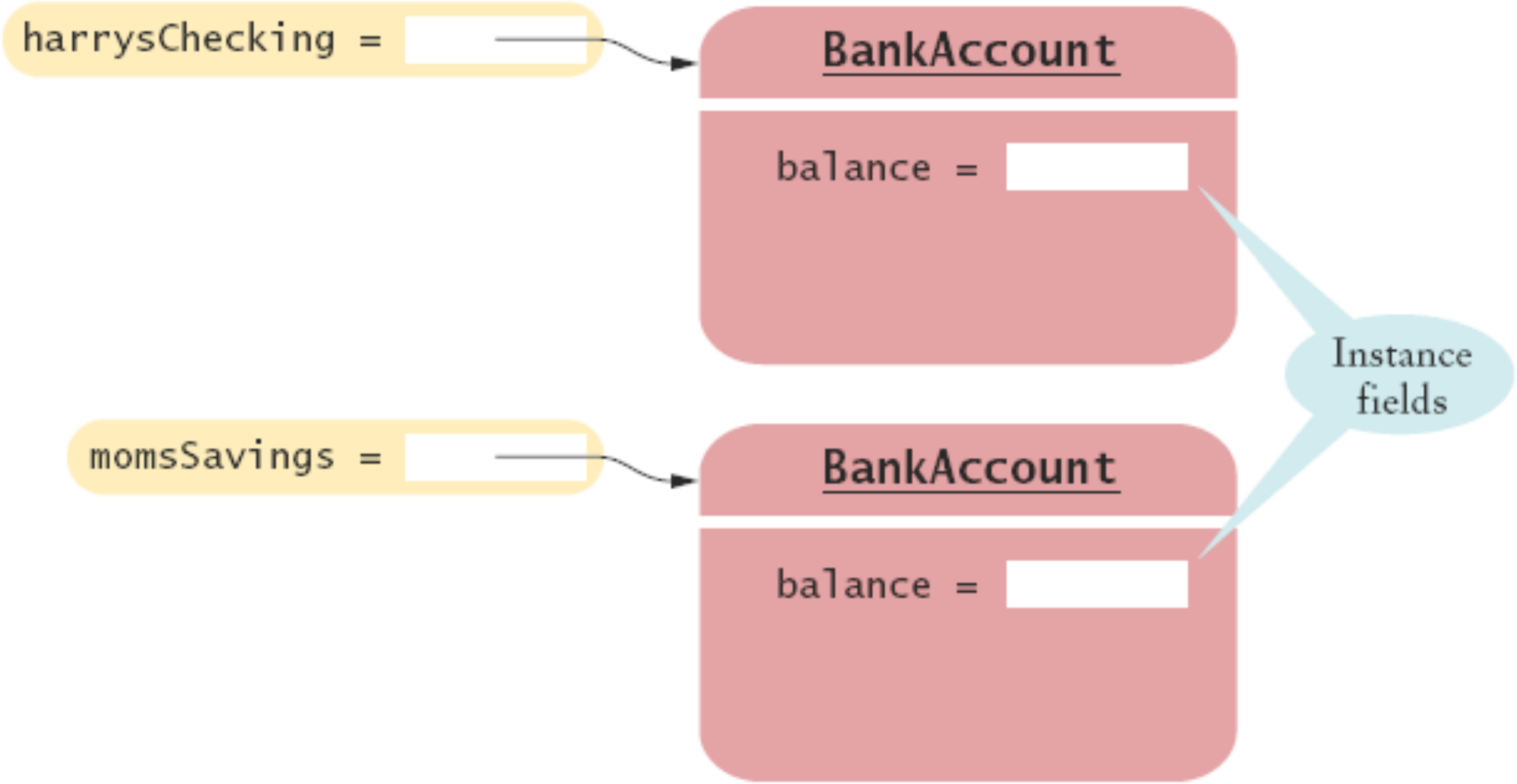


Figure 5 Instance Fields

Big Java by Cay Horstmann

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

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

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

```

Continued

ch03/account/BankAccount.java (cont.)

```java
   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      double newBalance = balance + amount;
      balance = newBalance;
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      double newBalance = balance - amount;
      balance = newBalance;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
```

Continued

ch03/account/BankAccount.java (cont.)

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

   private double balance;
}
```

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;
}
```

Big Java by Cay Horstmann

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

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*

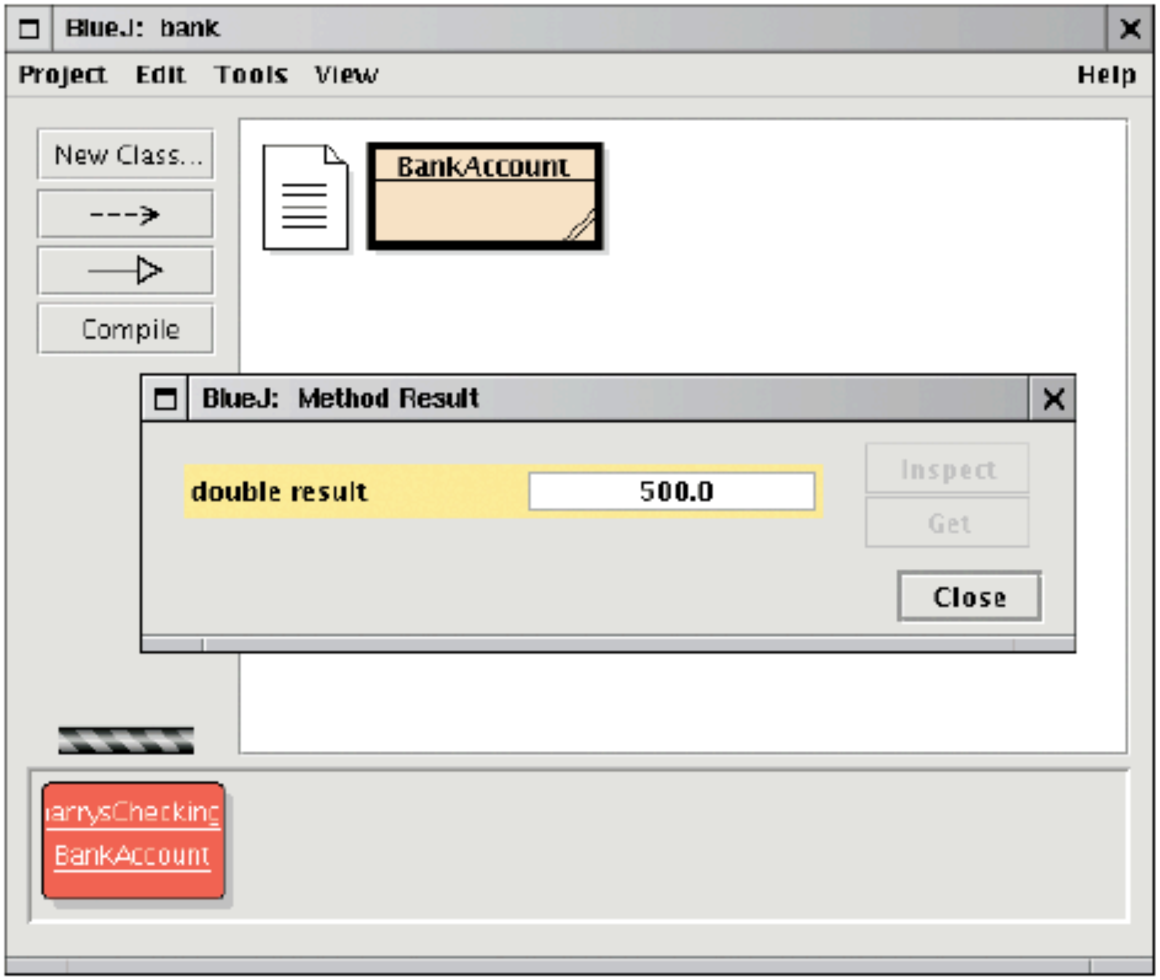
ch03/account/BankAccountTester.java

```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");
   }
}
```

Output:

```
1500
Expected: 1500
```

Testing With BlueJ



double result 500.0

Figure 6
The Return Value of the getBalance Method in BlueJ

Big Java by Cay Horstmann

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

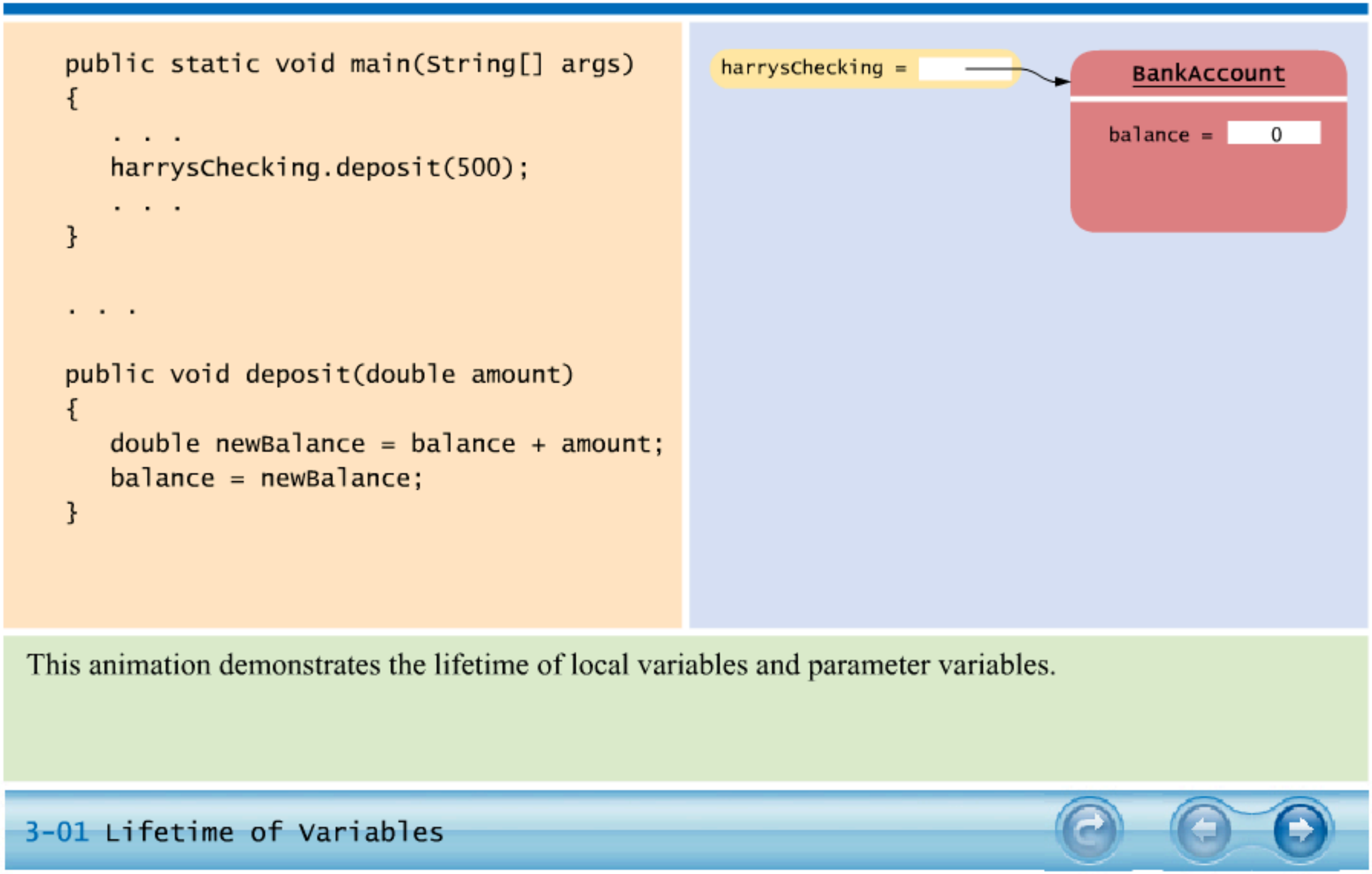
Why is the `BankAccountTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

> **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 –

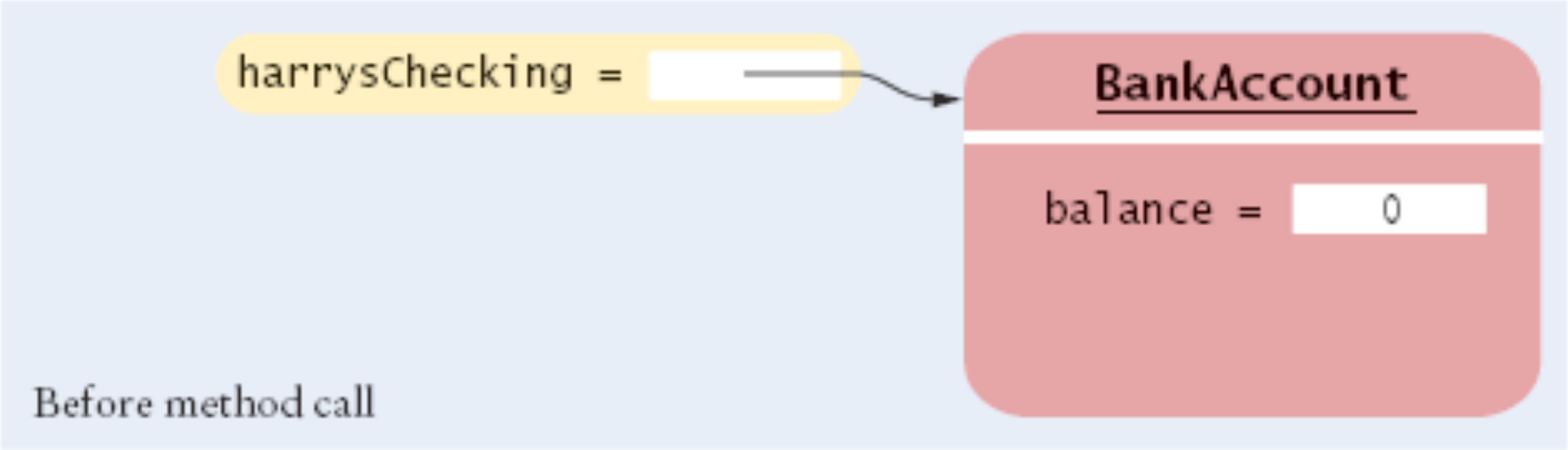


```
public static void main(String[] args)
{
   . . .
   harrysChecking.deposit(500);
   . . .
}

. . .

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`

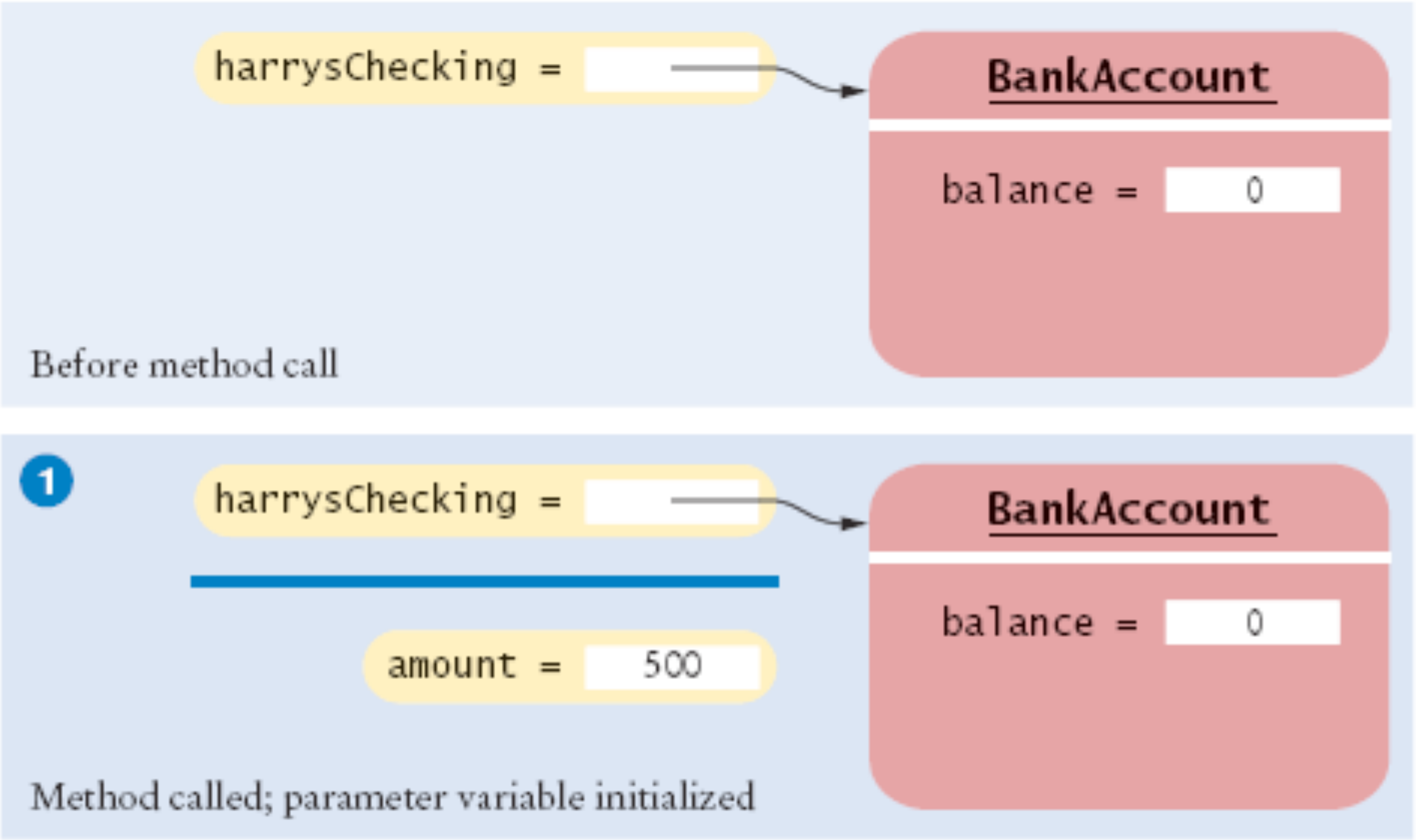
```
harrysChecking.deposit(500);
```



Big Java by Cay Horstmann

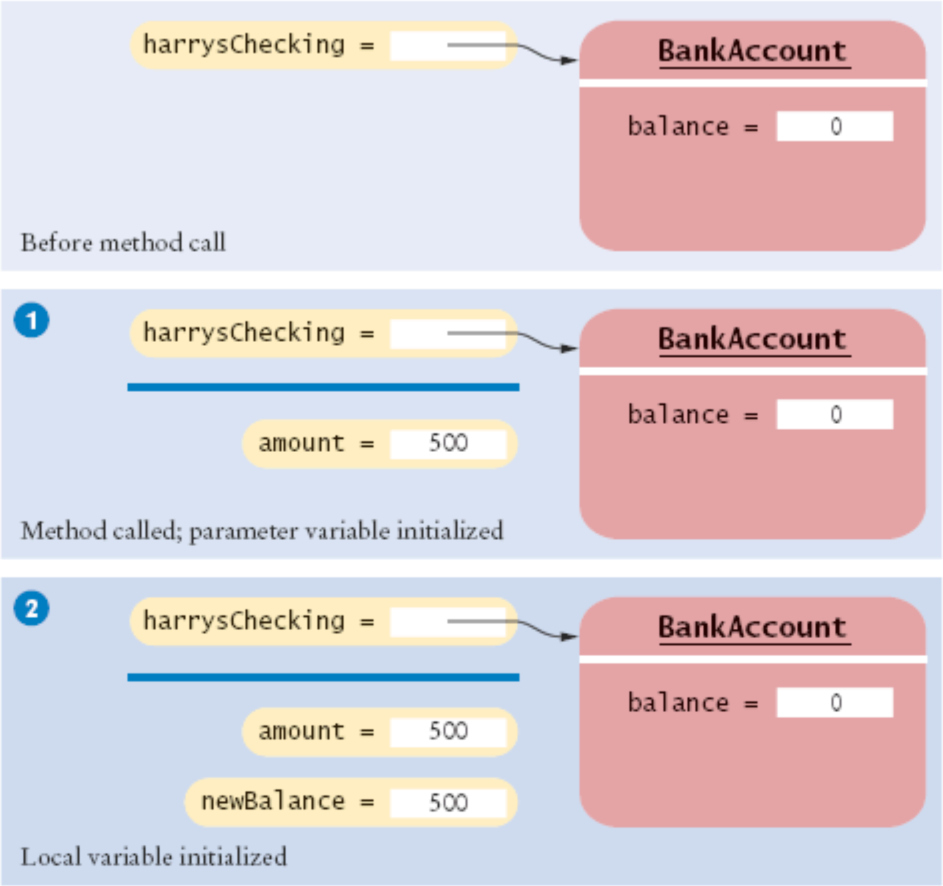
Lifetime of Variables – Calling Method `deposit`

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



Lifetime of Variables – Calling Method `deposit`

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



Big Java by Cay Horstmann

Lifetime of Variables – Calling Method `deposit`

```
harrysChecking.deposit(500); ❶
double newBalance = balance + amount; ❷
balance = newBalance; ❸
```

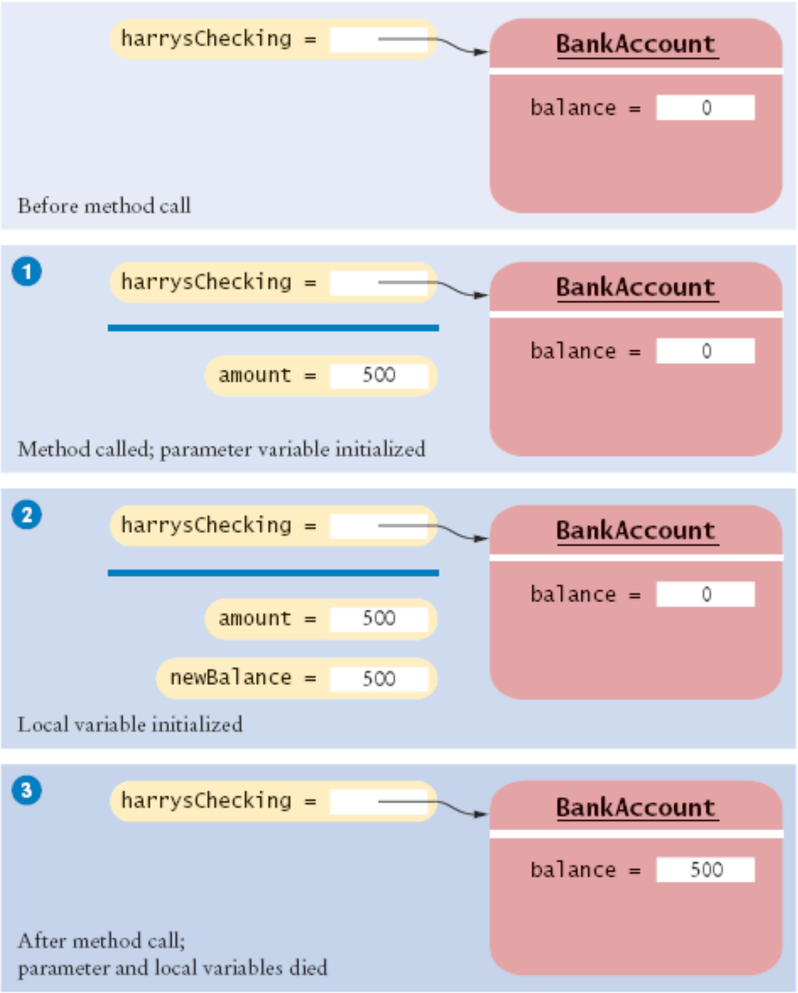


Figure 7 Lifetime of Variables

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).

Big Java by Cay Horstmann

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`

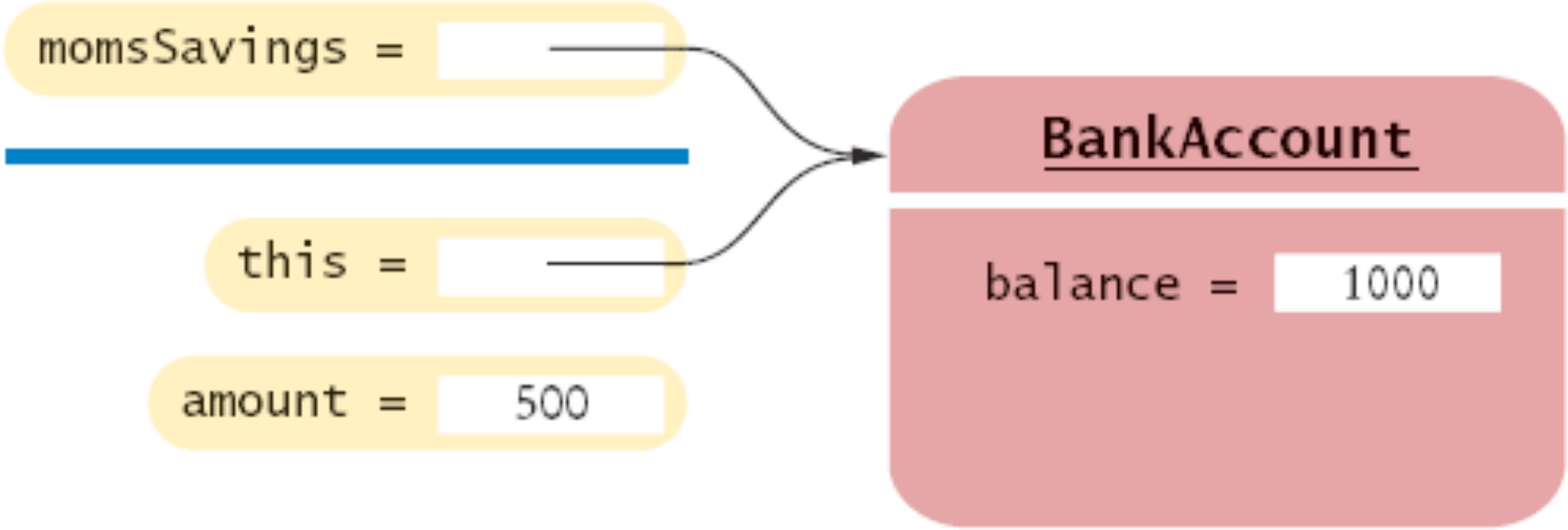


Figure 8 The Implicit Parameter of a Method Call

Big Java by Cay Horstmann

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`.

Big Java by Cay Horstmann

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

Drawing Cars (cont.)

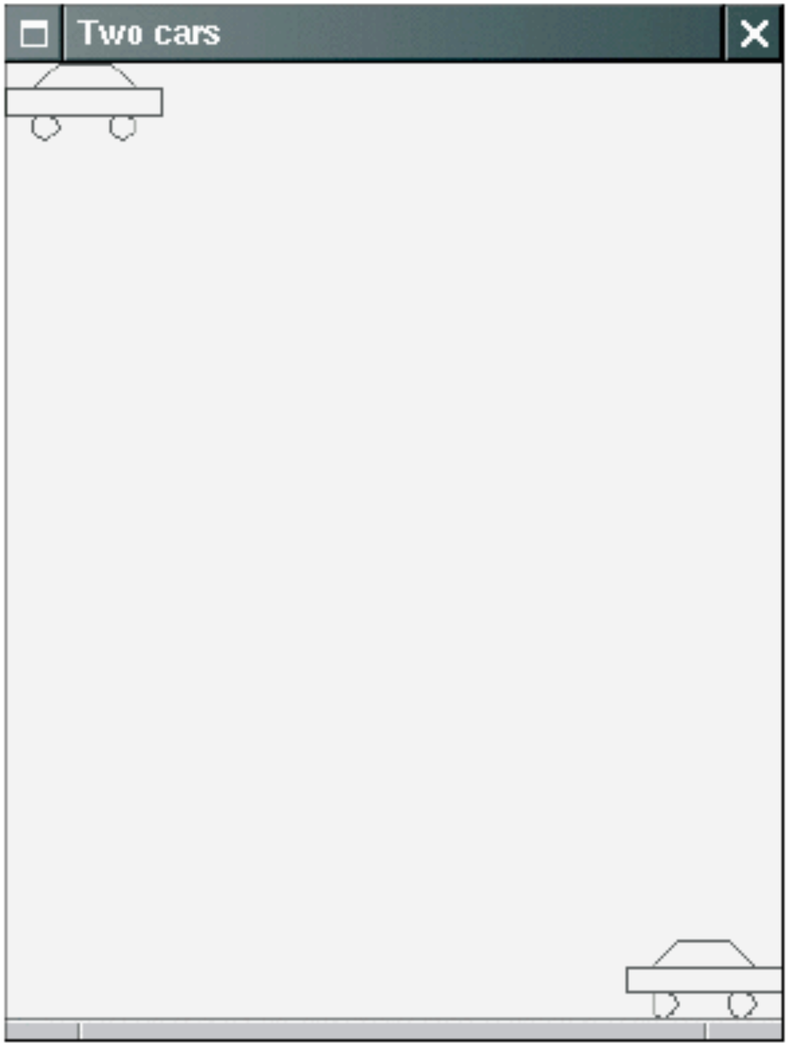


Figure 9
The Car Component Draws Two Car Shapes

Big Java by Cay Horstmann

Plan Complex Shapes on Graph Paper

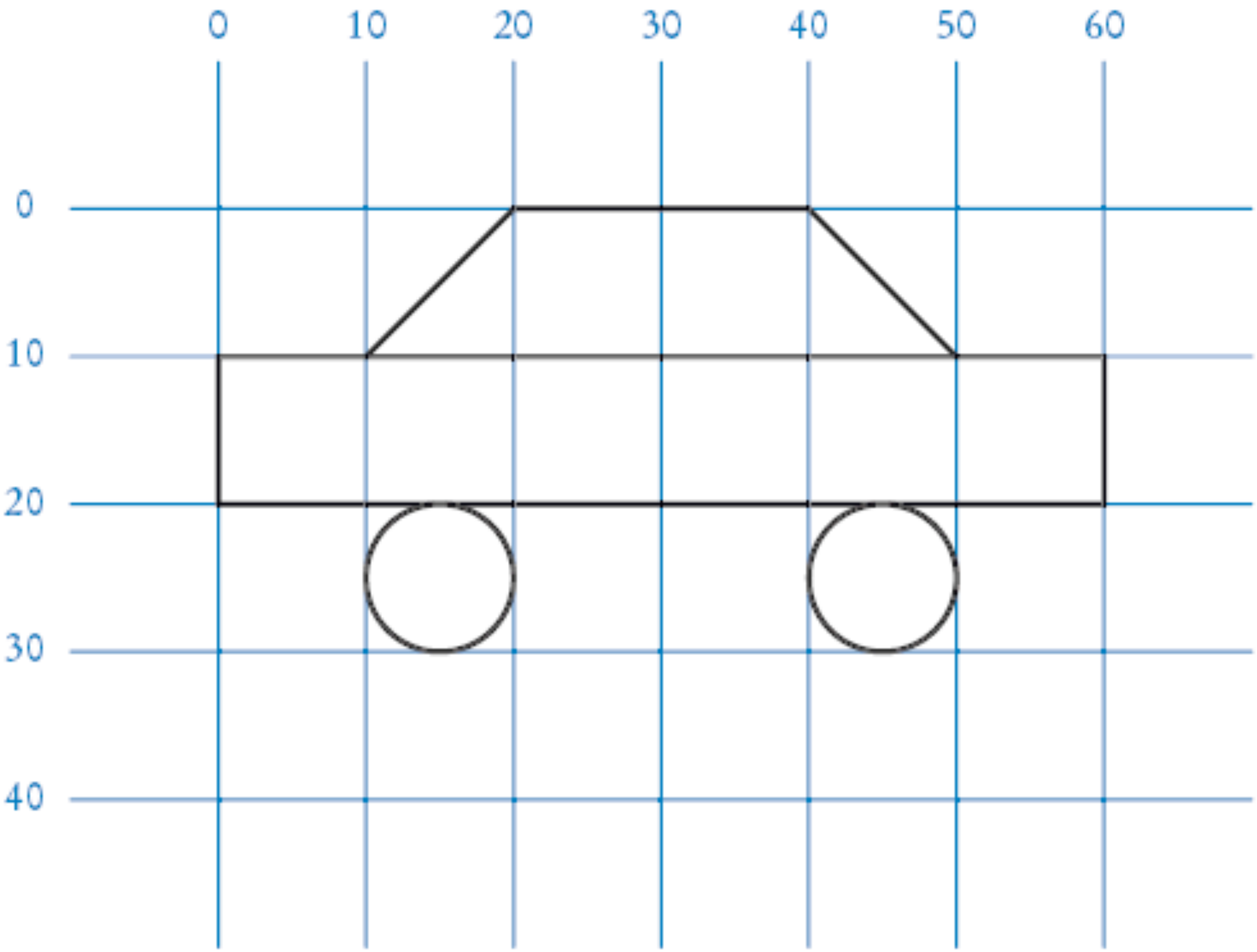


Figure 10 Using Graph Paper to Find Shape Coordinates

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

```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
{
   /**
      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;
   }

```

Continued

ch03/car/Car.java (cont.)

```java
   /**
      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
            = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);

      // The bottom of the front windshield
      Point2D.Double r1
            = new Point2D.Double(xLeft + 10, yTop + 10);
      // The front of the roof
      Point2D.Double r2
            = new Point2D.Double(xLeft + 20, yTop);
      // The rear of the roof
      Point2D.Double r3
            = new Point2D.Double(xLeft + 40, yTop);
      // The bottom of the rear windshield
```

Continued

ch03/car/Car.java (cont.)

```
      Point2D.Double r4
            = new Point2D.Double(xLeft + 50, yTop + 10);

      Line2D.Double frontWindshield
            = new Line2D.Double(r1, r2);
      Line2D.Double roofTop
            = new Line2D.Double(r2, r3);
      Line2D.Double rearWindshield
            = new Line2D.Double(r3, r4);

      g2.draw(body);
      g2.draw(frontTire);
      g2.draw(rearTire);
      g2.draw(frontWindshield);
      g2.draw(roofTop);
      g2.draw(rearWindshield);
   }

   private int xLeft;
   private int yTop;
}
```

ch03/car/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);
   }
}
```

ch03/car/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);
   }
}
```

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);
```

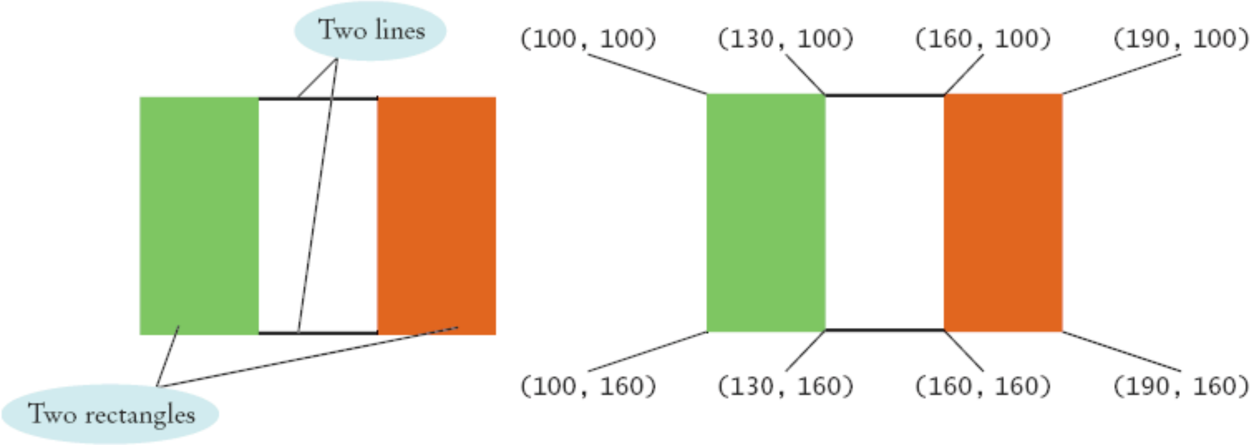
Big Java by Cay Horstmann

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);
```

Big Java by Cay Horstmann