Day 02

BASIC CONCEPTS OF JAVA 2

FUNDAMENTALS OF TELECOMMUNICATIONS LAB

Dr. Huy Nguyen

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

BASIC CONCEPTS OF JAVA 2

OBJECTS (cont.)

Applets

- **Applet:** program that runs inside a web browser
- To implement an applet, use this code outline:

```
public class MyApplet extends JApplet
{
   public void paint(Graphics g)
   {
      // Recover Graphics2D
      Graphics2D g2 = (Graphics2D) g;
      // Drawing instructions go here
      . . .
   }
}
```

Applets

- This is almost the same outline as for a component, with two minor differences:
  1. *You extend `JApplet`, not `JComponent`*
  2. *You place the drawing code inside the `paint` method, not inside `paintComponent`*
- To run an applet, you need an HTML file with the `applet` tag
- An HTML file can have multiple applets; add a separate `applet` tag for each applet
- You view applets with the applet viewer or a Java enabled browser:

```
appletviewer RectangleApplet.html
```

RectangleApplet.java

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JApplet;

/**
   An applet that draws two rectangles.
*/
public class RectangleApplet extends JApplet
{
   public void paint(Graphics g)
   {
      // Prepare for extended graphics
      Graphics2D g2 = (Graphics2D) g;

      // Construct a rectangle and draw it
      Rectangle box = new Rectangle(5, 10, 20, 30);
      g2.draw(box);

      // Move rectangle 15 units to the right and 25 units down
      box.translate(15, 25);

      // Draw moved rectangle
      g2.draw(box);
   }
}
```

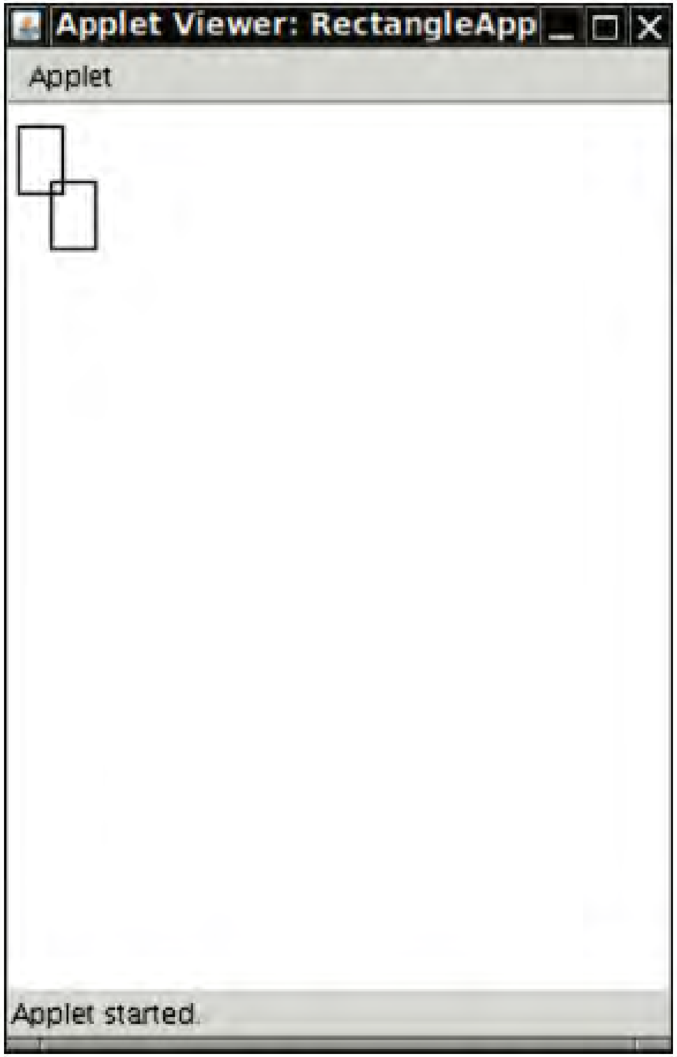
RectangleApplet.html

```
<applet code="RectangleApplet.class" width="300" height="400">
</applet>
```

RectangleAppletExplained.html

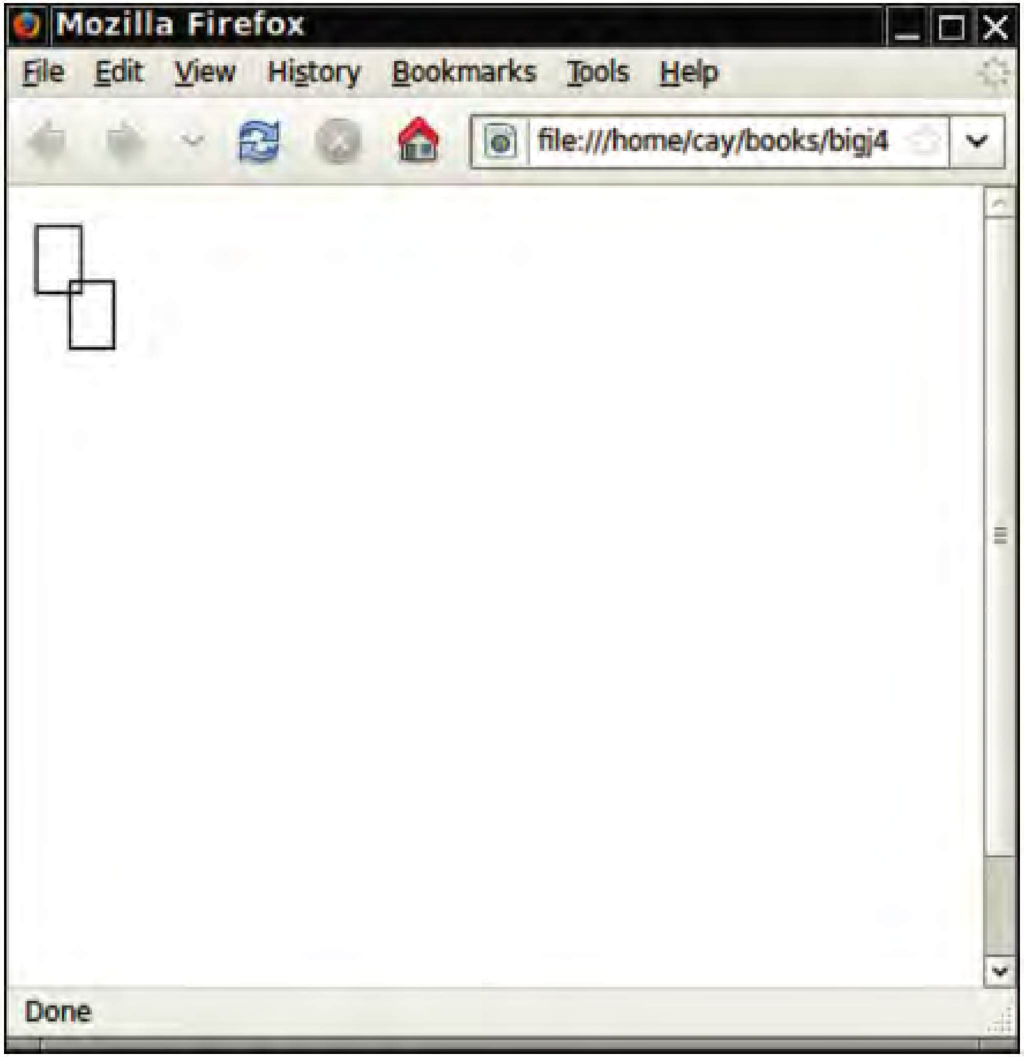
```
<html>
   <head>
      <title>Two rectangles</title>
   </head>
   <body>
      <p>Here is my <i>first applet</i>:</p>
      <applet code="RectangleApplet.class" width="300" height="400">
      </applet>
   </body>
</html>
```

Applets



Applet started.

An Applet in the Applet Viewer



Done

An Applet in a Web Browser

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

BASIC CONCEPTS OF JAVA 2

CLASSES

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 variables and local variables
- To be able to write javadoc comments
- To implement classes for drawing graphical shapes

Instance Variables

- **Example:** tally counter
- Simulator statements:

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

- Each counter needs to store a variable that keeps track of how many times the counter has been advanced

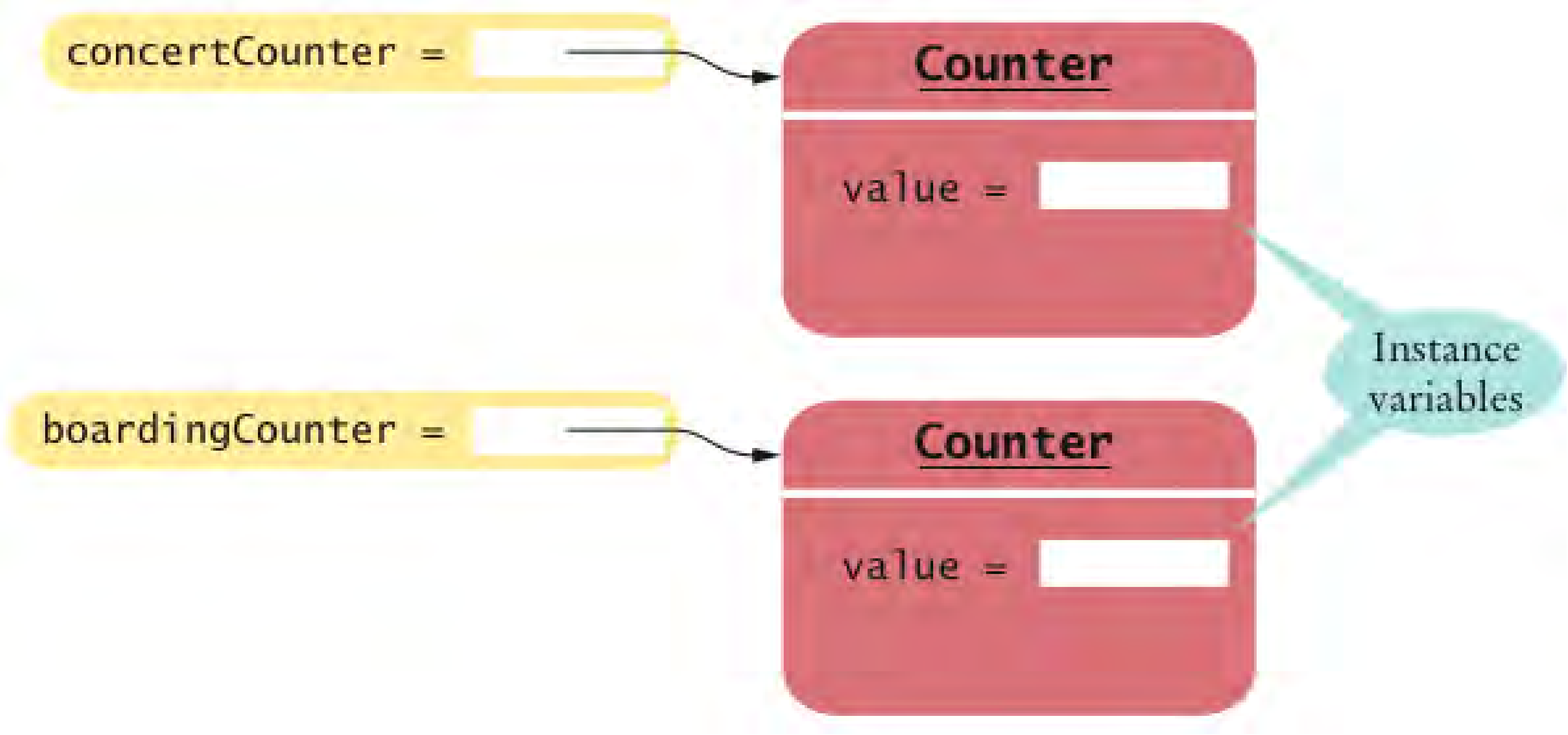
Instance Variables

- **Instance variables** store the data of an object
- **Instance of a class:** an object of the class
- The class declaration

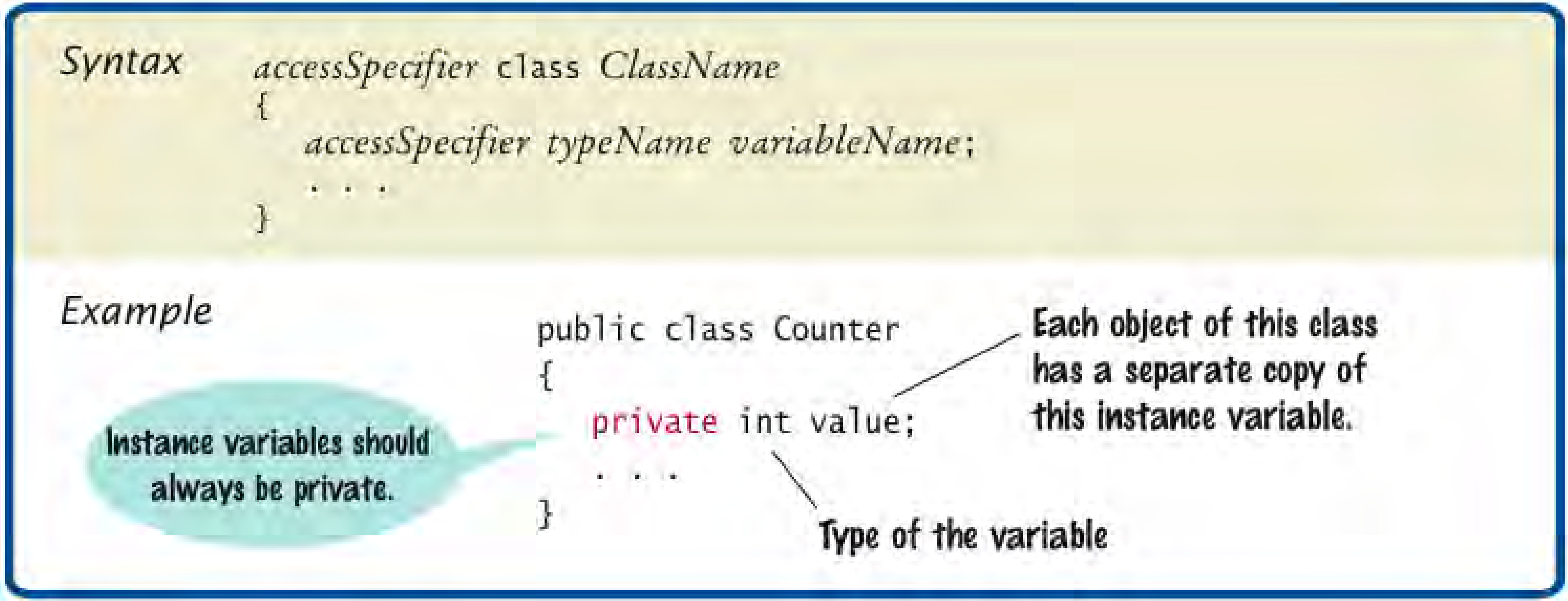
```
public class Counter
{
   private int value;
   …
}
```

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

Instance Variables



Syntax Instance Variable Declaration



Syntax

```
accessSpecifier class ClassName
{
   accessSpecifier typeName variableName;
   . . .
}
```

Example

```
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

Accessing Instance Variables

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

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

- The `getValue` method returns the current value:

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

- Private instance variables can only be accessed by methods of the same class

Self Check

Supply the body of a method `public void reset()` that resets the counter back to zero.

Answer:

```
public void reset()
{
   value = 0;
}
```

Self Check

Suppose you use a class `Clock` with private instance variables `hours` and `minutes`. How can you access these variables in your program?

Instance Variables

- **Encapsulation** is the process of hiding object data 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

Self Check

Consider the `Counter` class. A counter’s value starts at 0 and is advanced by the `count` method, so it should never be negative. Suppose you found a negative `value` variable during testing. Where would you look for the error?

Self Check

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

Self Check

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?

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 Declaration

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() { . . . }`

Specifying the Public Interface of a Class: Method Header

- access specifier (such as `public`)
- return type (such as `void` or `double`)
- method name (such as `deposit`)
- list of parameter variables (such as `double amount`)

Examples:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance()`

Specifying the Public Interface of a Class: Constructor Declaration

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

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

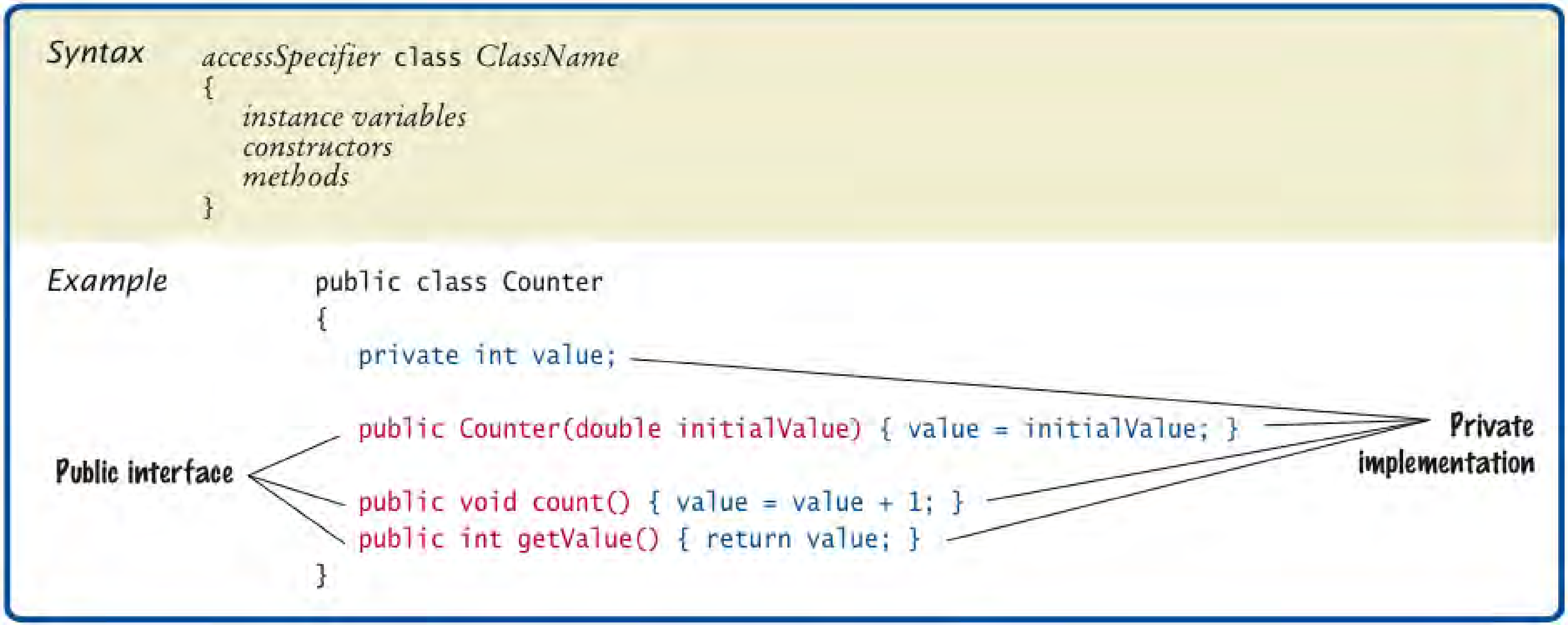
- Constructor body is executed when new object is created
- Statements in constructor body will set the internal data of the object that is being constructed
- All constructors of a class have the same name
- Compiler can tell constructors apart because they take different parameters

BankAccount Public Interface

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

```
public class BankAccount
{
   // private 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
   }
}
```

Syntax Class Declaration



Syntax

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

Example

```
public class Counter
{
    private int value;

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

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

Public interface
Private implementation

Self Check

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

Answer:

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

Self Check

What is wrong with this sequence of statements?

```
BankAccount harrysChecking = new BankAccount(10000);
System.out.println(harrysChecking.withdraw(500));
```

Self Check

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

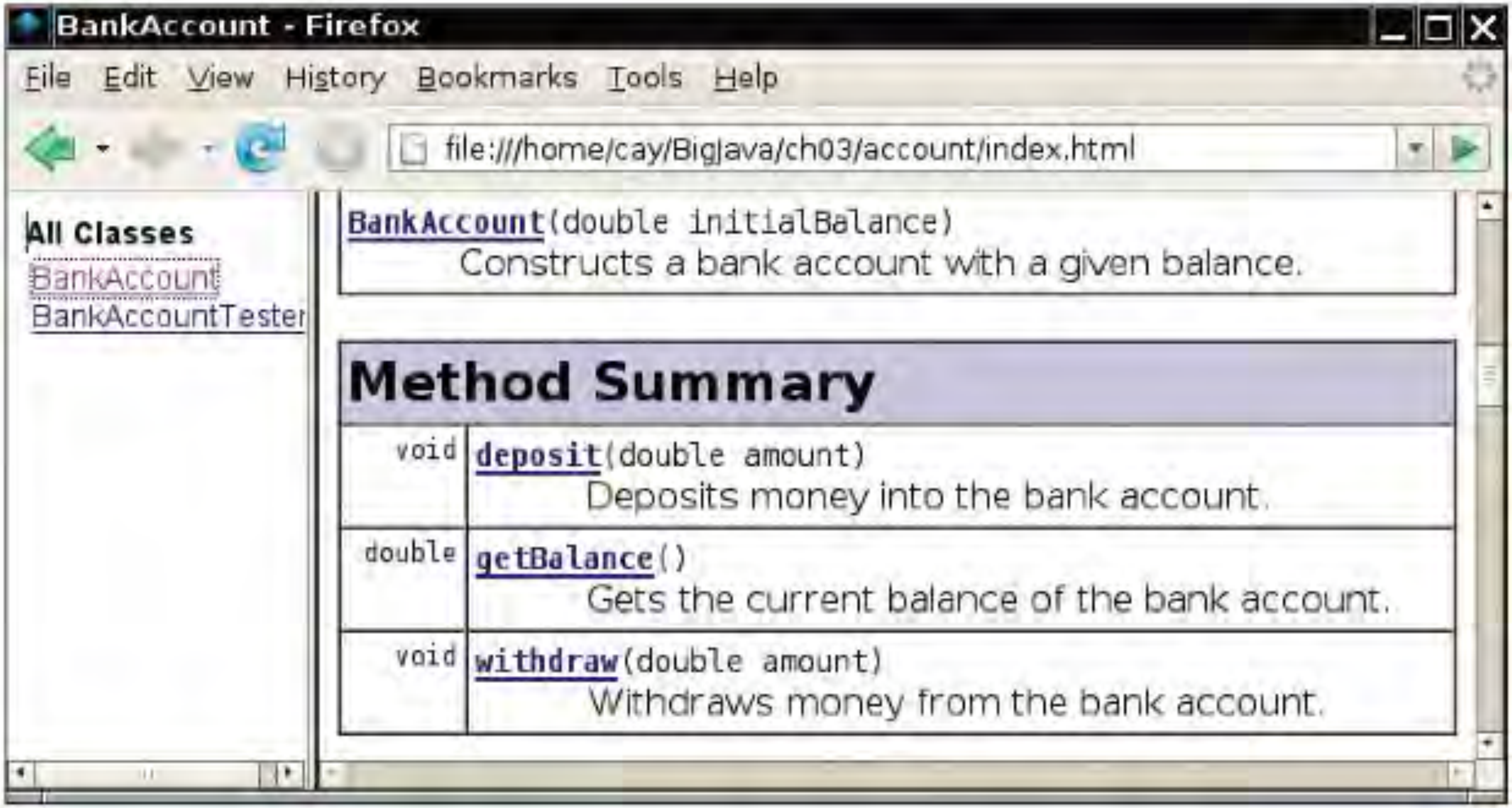
```
/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
   //implementation filled in later
}
/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   //implementation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



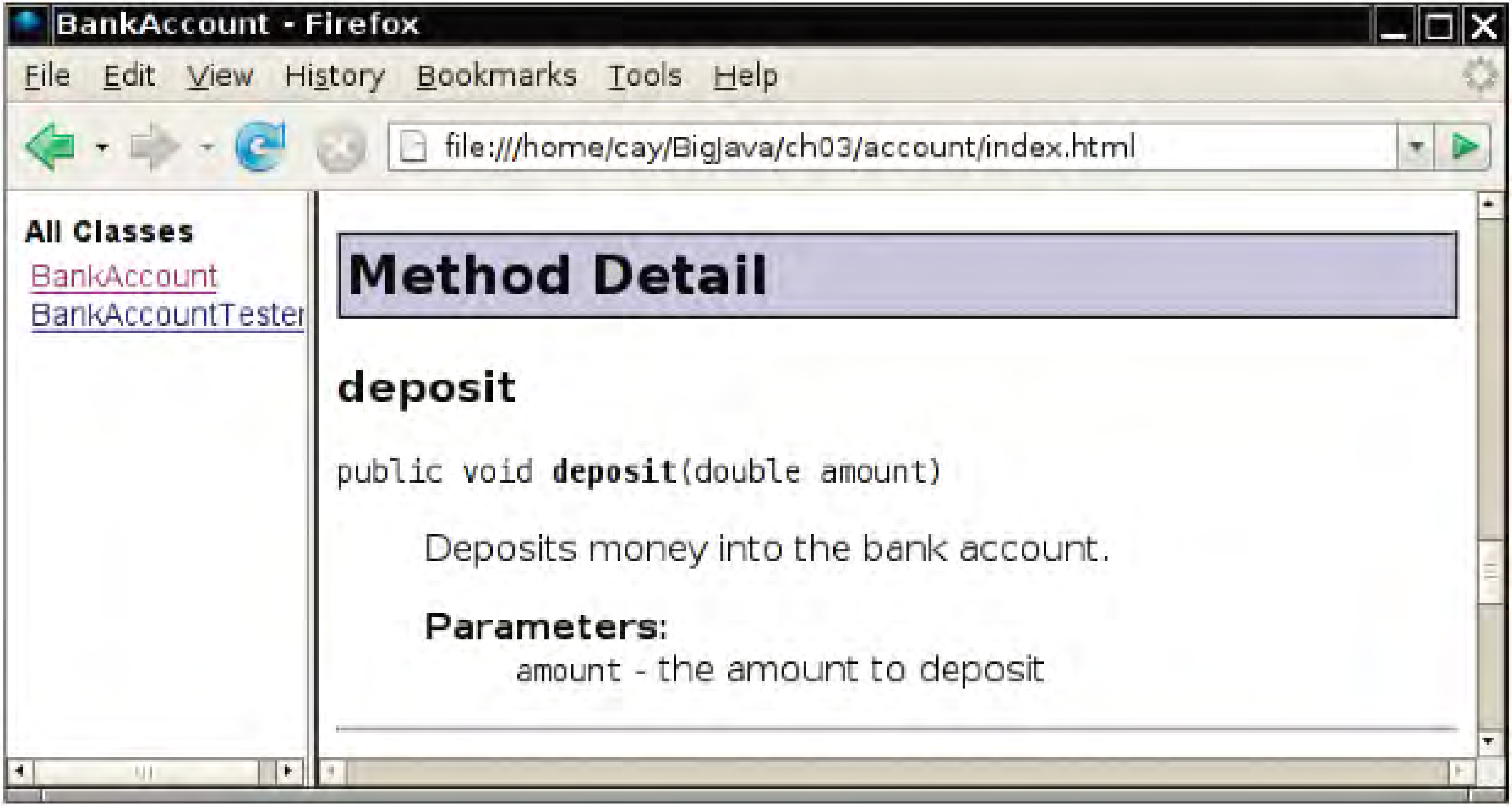
BankAccount - Firefox
File Edit View History Bookmarks Tools Help
file:///home/cay/BigJava/ch03/account/index.html
All Classes
BankAccount
BankAccountTester

BankAccount(double initialBalance)
Constructs a bank account with a given balance.

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.

Javadoc Method Detail


BankAccount - Firefox
File Edit View History Bookmarks Tools Help
file:///home/cay/BigJava/ch03/account/index.html
All Classes
BankAccount
BankAccountTester

Method Detail

deposit

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

Deposits money into the bank account.

Parameters:
amount - the amount to deposit

Self Check

Provide documentation comments for the `Counter` class

Answer:

```
/**
   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 count()
   {
      value = value + 1;
   }
}
```

Self Check

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

Why is the following documentation comment questionable?

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

Implementing Constructors

- Constructors contain instructions to initialize the instance variables of an object:

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

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

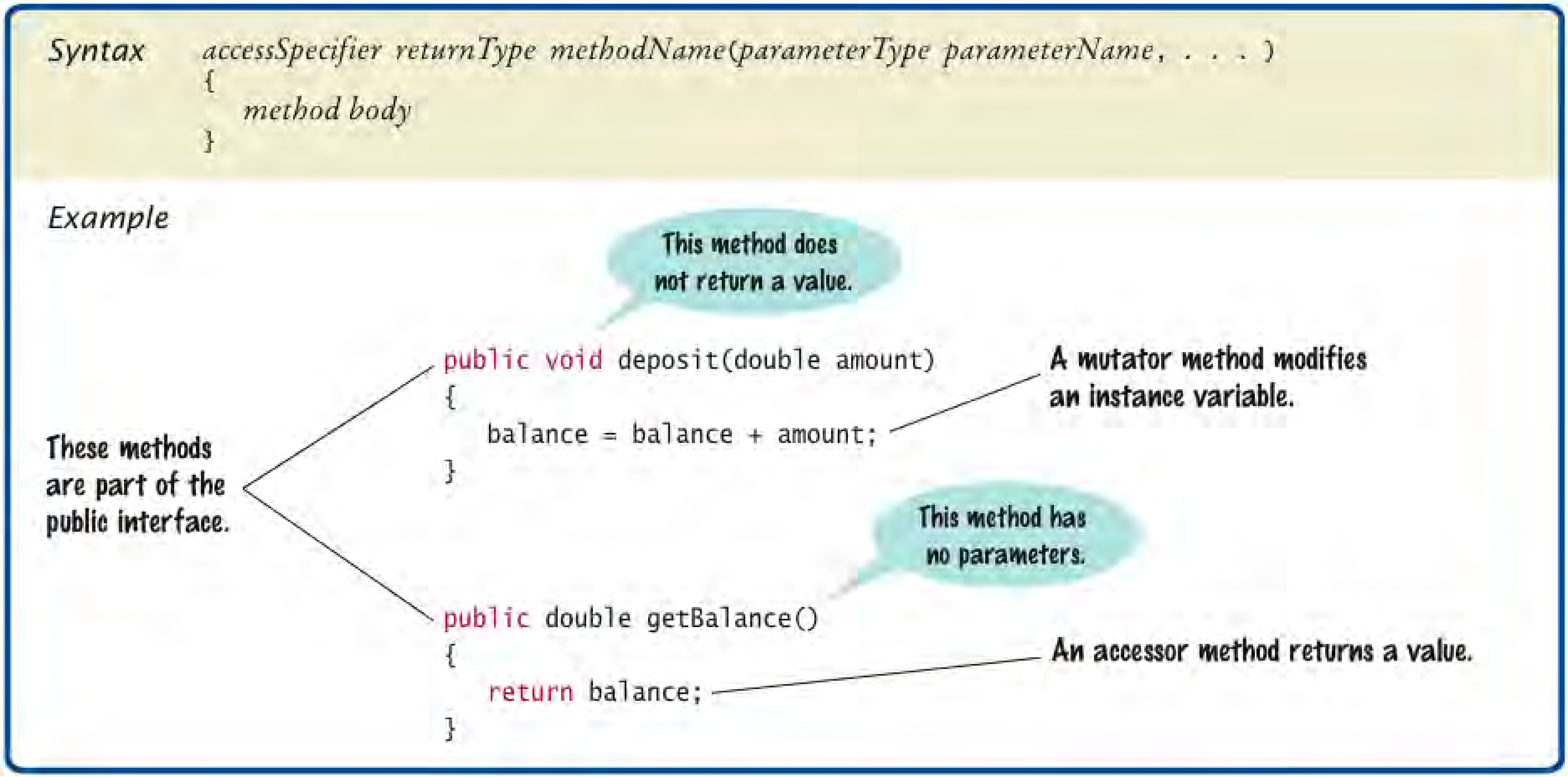
Constructor Call Example

- Statement:

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

  - *Create a new object of type `BankAccount`*
  - *Call the second constructor (because a construction parameter is supplied in the constructor call)*
  - *Set the parameter variable `initialBalance` to `1000`*
  - *Set the `balance` instance variable 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*

Syntax Method Declaration



Syntax

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

Example

This method does not return a value.

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

A mutator method modifies an instance variable.

These methods are part of the public interface.

This method has no parameters.

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

An accessor method returns a value.

Implementing Methods

- `deposit` method:

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

Method Call Example

- Statement:

  `harrysChecking.deposit(500);`

  - *Set the parameter variable `amount` to 500*
  - *Fetch the `balance` variable of the object whose location is stored in `harrysChecking`*
  - *Add the value of `amount` to `balance`*
  - *Store the sum in the `balance` instance variable, overwriting the old value*

Implementing Methods

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

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

BankAccount.java

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

   /**
      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;
   }
```

BankAccount.java (cont.)

```

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

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

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

Self Check

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

Answer: An instance variable

```
private int accountNumber;
```

needs to be added to the class.

Self Check

Why does the following code not succeed in robbing mom’s bank account?

```
public class BankRobber
{
   public static void main(String[] args)
   {
      BankAccount momsSavings = new BankAccount(1000);
      momsSavings.balance = 0;
   }
}
```

Answer: Because the `balance` instance variable is accessed from the `main` method of `BankRobber`. The compiler will report an error because `balance` has private access in `BankAccount`.

Self Check

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

Answer:

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

Self Check

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
- *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*

BankAccountTester.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 (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*

Self Check

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

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.

Local Variables

- Local and parameter variables belong to a method
  - *When a method or constructor runs, its local and parameter variables come to life*
  - *When the method or constructor exits, they are removed immediately*
- Instance variables belongs to an 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*
- In Java, the *garbage collector* periodically reclaims objects when they are no longer used
- Instance variables are initialized to a default value, but you must initialize local variables

Self Check

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

Why was it necessary to introduce the local variable `change` in the `giveChange` method? That is, why didn't the method simply end with the statement

```
   return payment - purchase;
```

Answer: After computing the change due, `payment` and `purchase` were set to zero. If the method returned `payment - purchase`, it would always return zero.

Implicit Parameter

- The **implicit parameter** of a method is the object on which the method is invoked

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

- In the call

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

  The implicit parameter is `momsSavings` and the explicit parameter is `500`

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

Implicit Parameters and `this`

- The `this` reference denotes the implicit parameter

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

  actually means

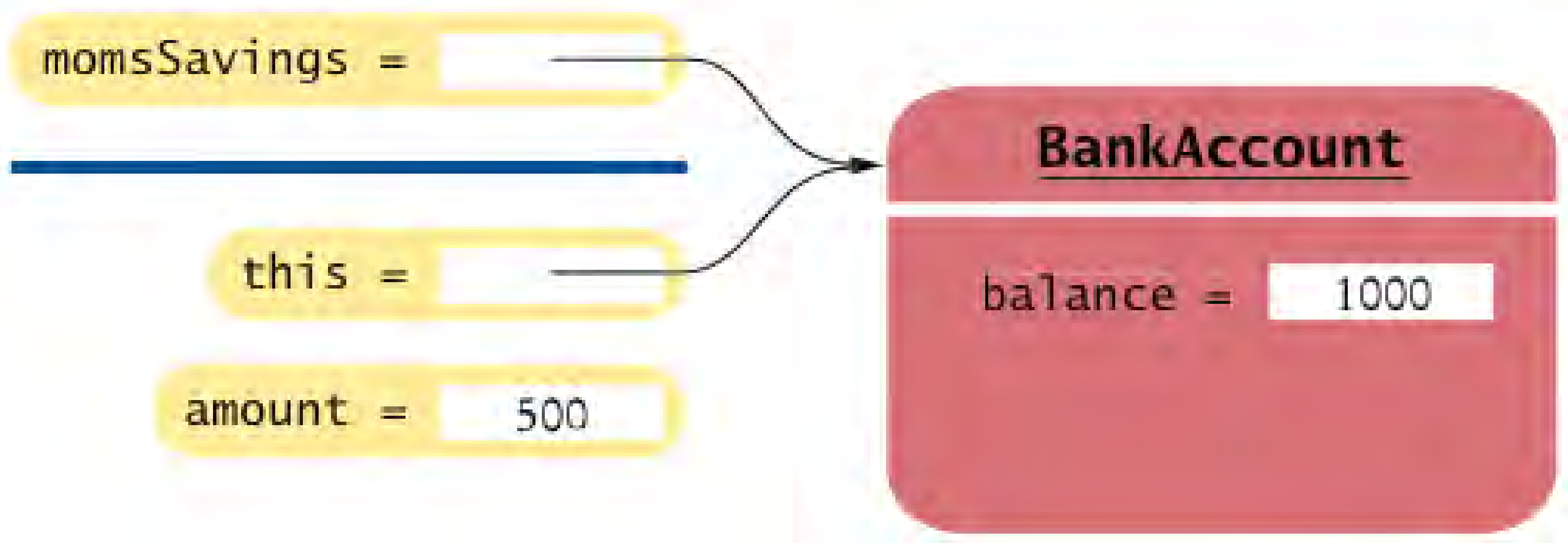
  ```
  this.balance = this.balance + amount;
  ```

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

Implicit Parameters and `this`

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

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



Implicit Parameters and `this`

- 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

Implicit Parameters and `this`

- 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
   }
}
```

Self Check

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

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 variable named `amount`.

Self Check

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 main method is not ivoked on any object - and one explicit parameter, called `args`.

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

BASIC CONCEPTS OF JAVA 2

DECISIONS

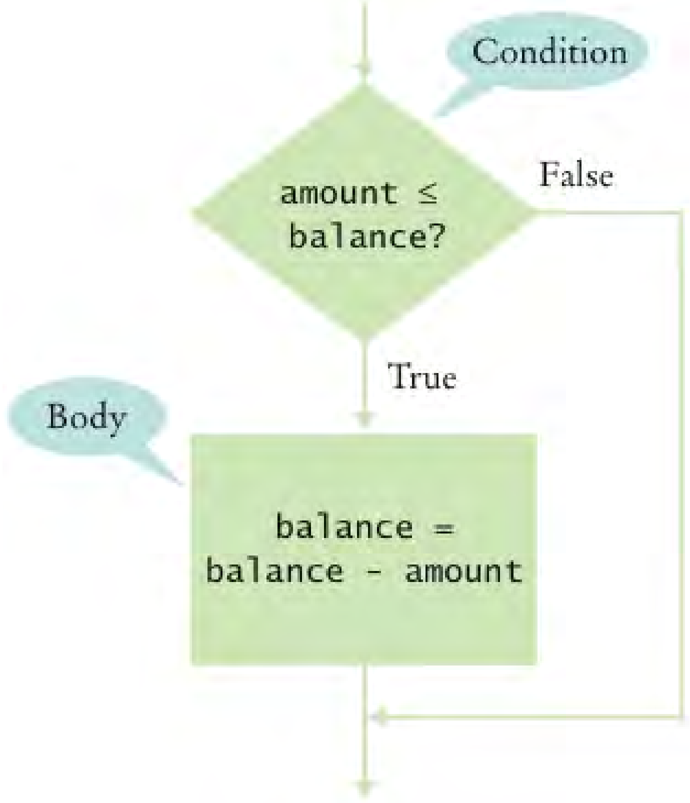
Chapter Goals

- To be able to implement decisions using `if` statements
- To understand how to group statements into blocks
- To learn how to compare integers, floating-point numbers, strings, and objects
- To recognize the correct ordering of decisions in multiple branches
- To program conditions using Boolean operators and variables
- To understand the importance of test coverage

The `if` Statement

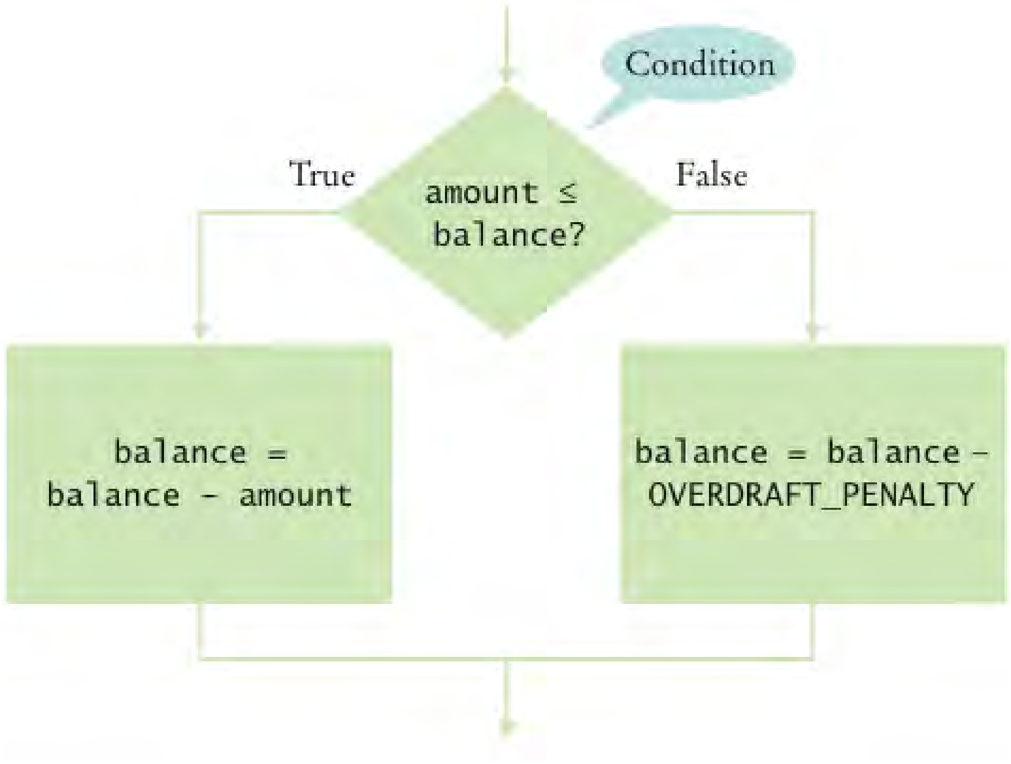
- The `if` statement lets a program carry out different actions depending on a condition

```
if (amount <= balance)
   balance = balance - amount;
```



The `if/else` Statement

```
if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY
```



Statement Types

- Simple statement:

```
balance = balance - amount;
```

- Compound statement:

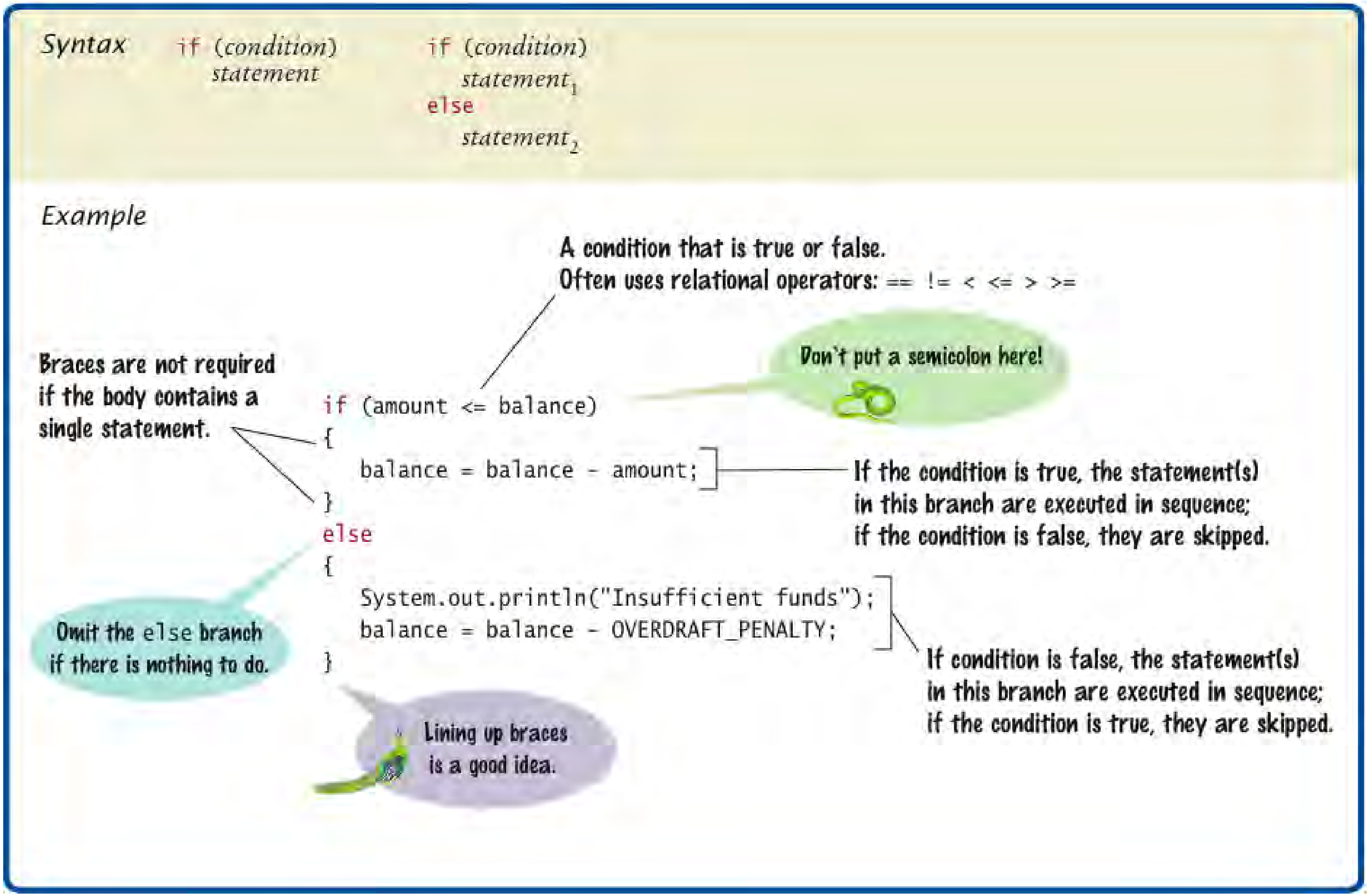
```
if (balance >= amount) balance = balance - amount;
```

Also loop statements in Iteration

- Block statement:

```
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Syntax The `if` Statement



Syntax

```
if (condition)
   statement
```

```
if (condition)
   statement1
else
   statement2
```

Example

```
if (amount <= balance)
{
   balance = balance - amount;
}
else
{
   System.out.println("Insufficient funds");
   balance = balance - OVERDRAFT_PENALTY;
}
```

A condition that is true or false.
Often uses relational operators: == != < <= > >=

Don't put a semicolon here!

Braces are not required if the body contains a single statement.

If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

Omit the `else` branch if there is nothing to do.

If condition is false, the statement(s) in this branch are executed in sequence; if the condition is true, they are skipped.

Lining up braces is a good idea.

Self Check

Why did we use the condition `amount <= balance` and not `amount < balance` in the example for the `if/else` statement?

Answer: If the withdrawal amount equals the balance, the result should be a zero balance and no penalty.

Self Check

What is logically wrong with the statement

```
if (amount <= balance)
   newBalance = balance - amount;
   balance = newBalance;
```

and how do you fix it?

Answer: Only the first assignment statement is part of the `if` statement. Use braces to group both assignment statements into a block statement.

Comparing Values: Relational Operators

- Relational operators compare values

Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
! =	≠	Not equal

Comparing Values: Relational Operators

- The == denotes equality testing:

```
a = 5; // Assign 5 to a
if (a == 5) ... // Test whether a equals 5
```

- Relational operators have lower precedence than arithmetic operators:

```
amount + fee <= balance
```

Comparing Floating-Point Numbers

- Consider this code:

```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
   System.out.println("sqrt(2)squared minus 2 is 0");
else
   System.out.println("sqrt(2)squared minus 2 is not 0 but "
      + d);
```

- It prints:

```
sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16
```

Comparing Floating-Point Numbers

- To avoid roundoff errors, don't use == to compare floating-point numbers
- To compare floating-point numbers test whether they are *close enough*: $|x - y| \le \varepsilon$

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
   // x is approximately equal to y
```

- ε is a small number such as 10^{-14}

Comparing Strings

- To test whether two strings are equal to each other, use `equals` method:

```
if (string1.equals(string2)) . . .
```

- Don’t use `==` for strings!

```
if (string1 == string2) // Not useful
```

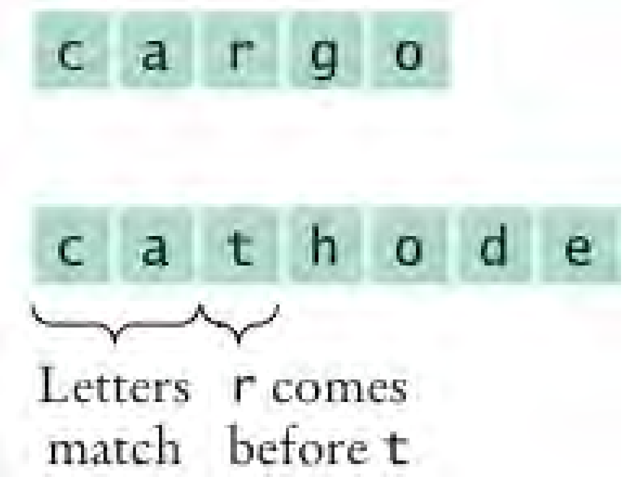
- `==` tests identity, `equals` tests equal contents
- Case insensitive test:

```
if (string1.equalsIgnoreCase(string2))
```

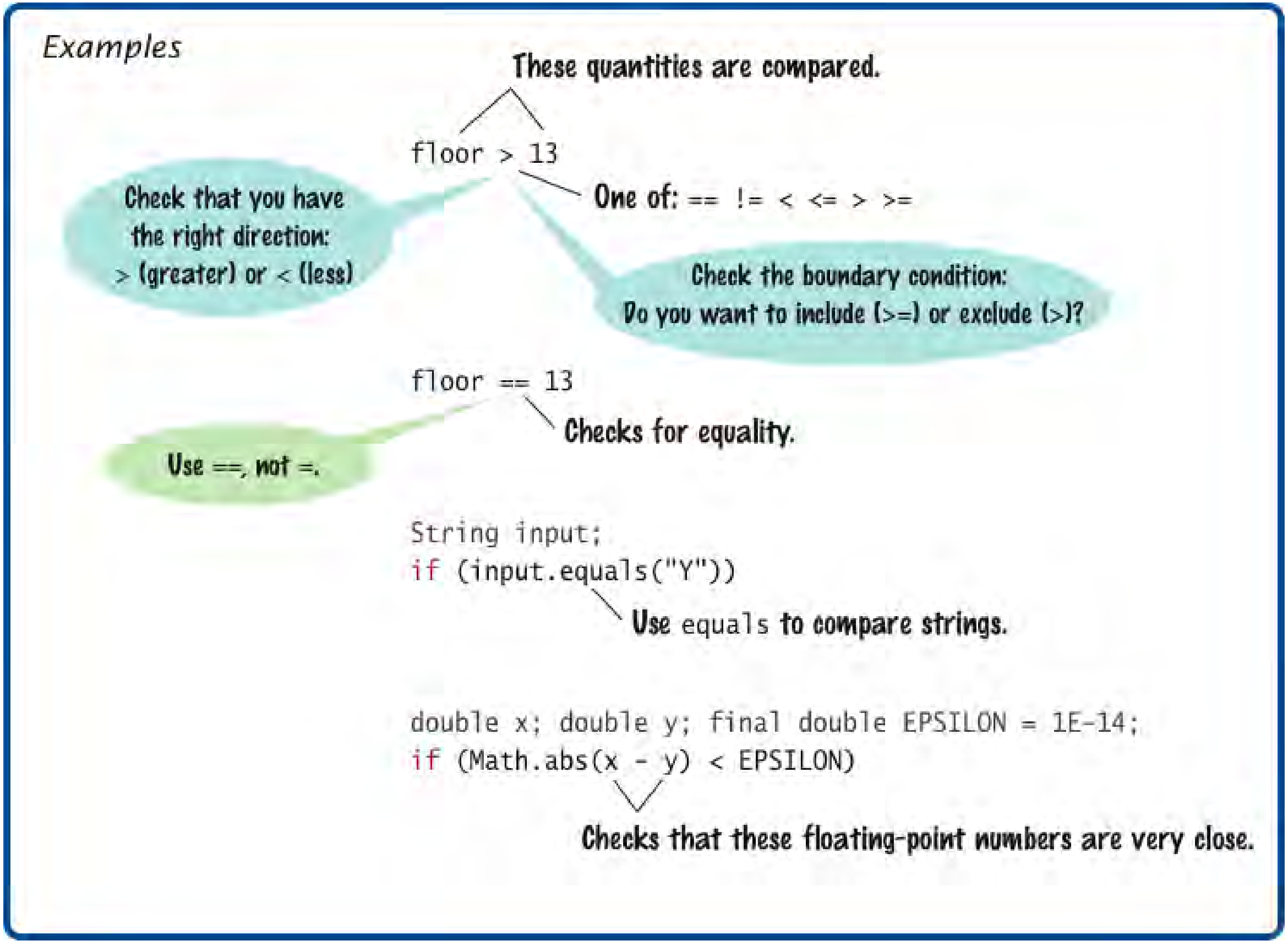
Comparing Strings

- `string1.compareTo(string2) < 0` means:
  `string1` comes before `string2` in the dictionary
- `string1.compareTo(string2) > 0` means:
  `string1` comes after `string2`
- `string1.compareTo(string2) == 0` means:
  `string1` equals `string2`
- `"car"` comes before `"cargo"`
- All uppercase letters come before lowercase:
  `"Hello"` comes before `"car"`

Lexicographic Comparison



Syntax Comparisons



Examples

These quantities are compared.

```
floor > 13
```

One of: == != < <= > >=

Check that you have the right direction: > (greater) or < (less)

Check the boundary condition: Do you want to include (>=) or exclude (>)?

```
floor == 13
```

Checks for equality.

Use ==, not =.

```
String input;
if (input.equals("Y"))
```

Use equals to compare strings.

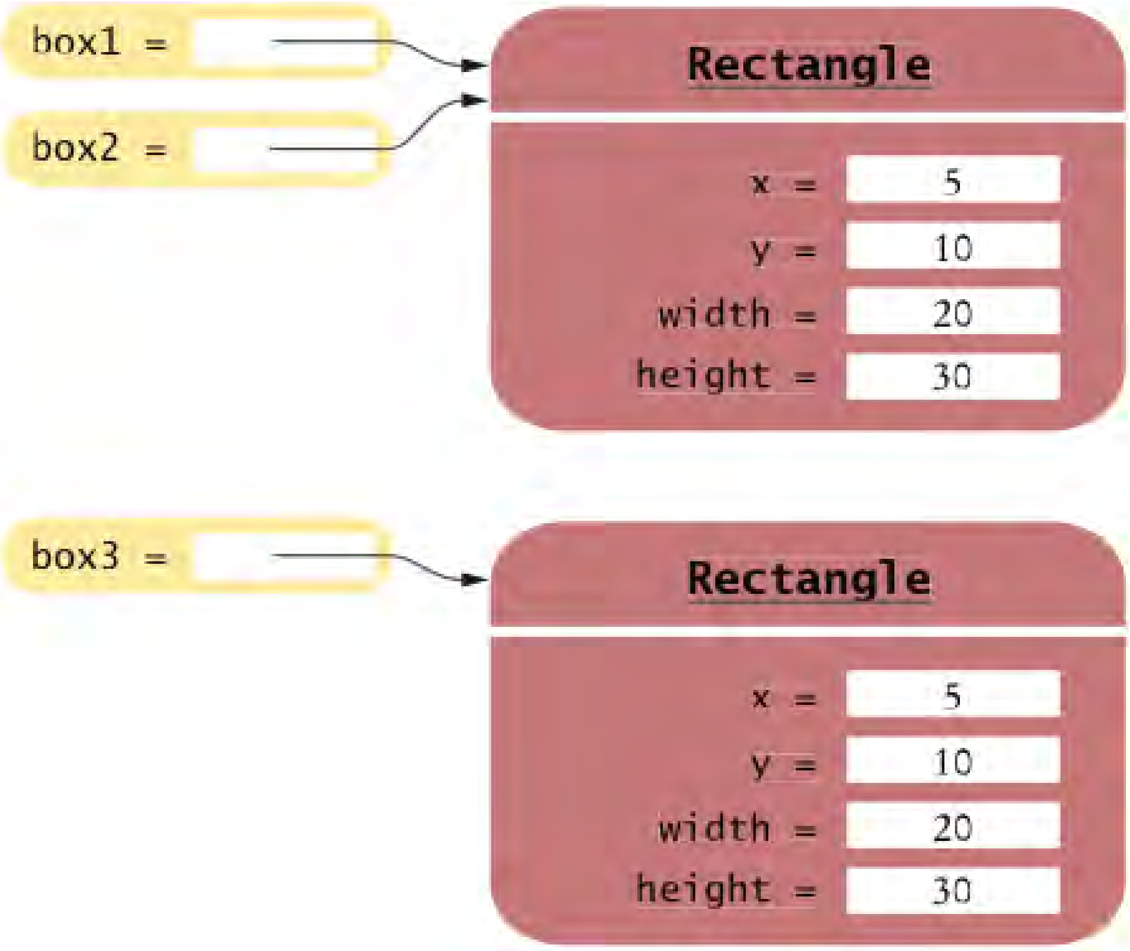
```
double x; double y; final double EPSILON = 1E-14;
if (Math.abs(x - y) < EPSILON)
```

Checks that these floating-point numbers are very close.

Comparing Objects

- `==` tests for identity, `equals` for identical content
-
  ```
  Rectangle box1 = new Rectangle(5, 10, 20, 30);
  Rectangle box2 = box1;
  Rectangle box3 = new Rectangle(5, 10, 20, 30);
  ```
- `box1 != box3`, but `box1.equals(box3)`
- `box1 == box2`
- Caveat: `equals` must be defined for the class

Object Comparison



Testing for `null`

- `null` reference refers to no object:

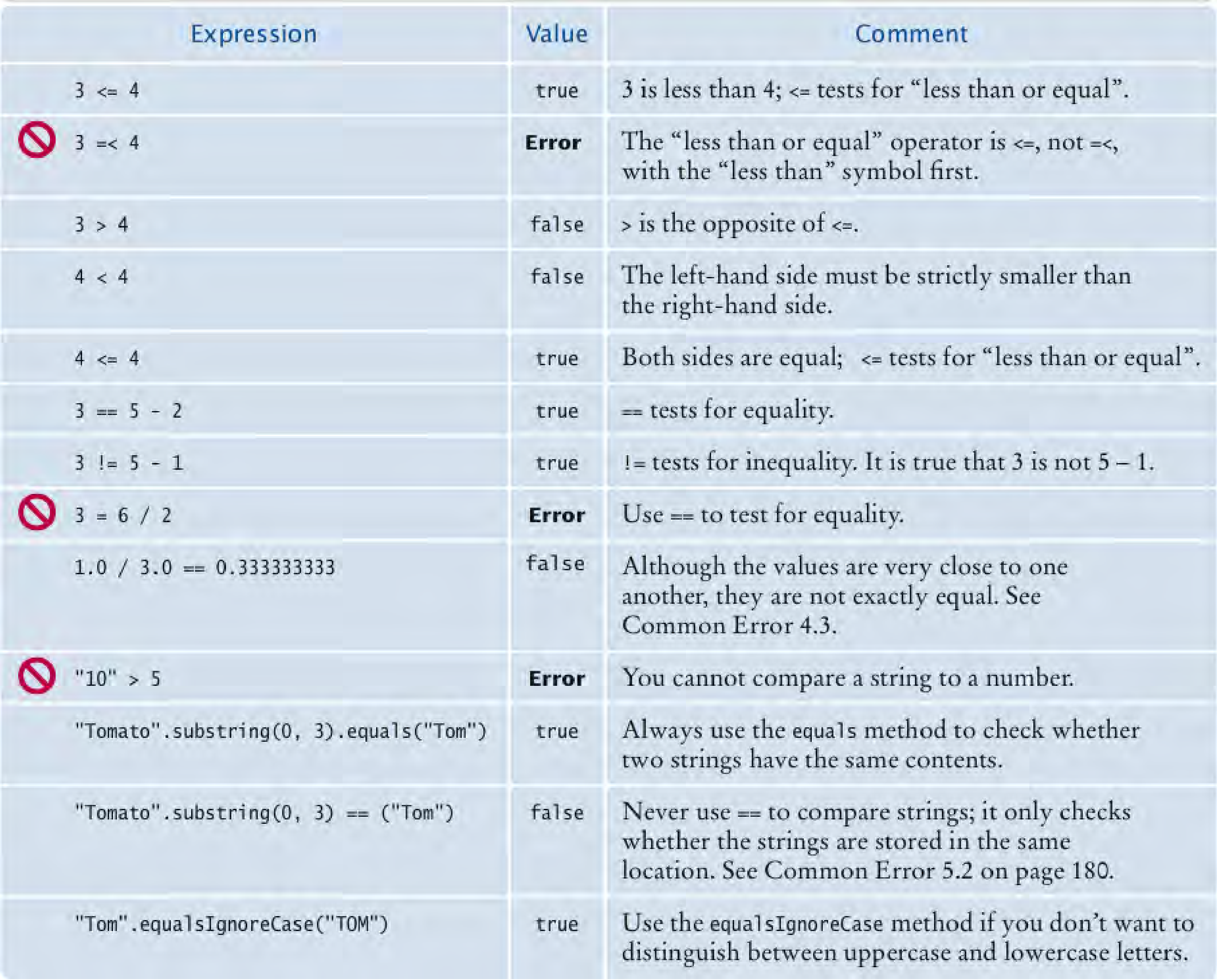
```
String middleInitial = null; // Not set
if ( ... )
   middleInitial = middleName.substring(0, 1);
```

- Can be used in tests:

```
if (middleInitial == null)
   System.out.println(firstName + " " + lastName);
else
   System.out.println(firstName + " " + middleInitial +
      ". " + lastName);
```

- Use `==`, not `equals`, to test for `null`
- `null` is not the same as the empty string `""`

Relational Operator Examples



	Expression	Value	Comment
	`3 <= 4`	`true`	3 is less than 4; <= tests for “less than or equal”.
🚫	`3 =< 4`	**Error**	The “less than or equal” operator is <=, not =<, with the “less than” symbol first.
	`3 > 4`	`false`	> is the opposite of <=.
	`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
	`4 <= 4`	`true`	Both sides are equal; <= tests for “less than or equal”.
	`3 == 5 - 2`	`true`	== tests for equality.
	`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
🚫	`3 = 6 / 2`	**Error**	Use == to test for equality.
	`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Common Error 4.3.
🚫	`"10" > 5`	**Error**	You cannot compare a string to a number.
	`"Tomato".substring(0, 3).equals("Tom")`	`true`	Always use the `equals` method to check whether two strings have the same contents.
	`"Tomato".substring(0, 3) == ("Tom")`	`false`	Never use == to compare strings; it only checks whether the strings are stored in the same location. See Common Error 5.2 on page 180.
	`"Tom".equalsIgnoreCase("TOM")`	`true`	Use the `equalsIgnoreCase` method if you don’t want to distinguish between uppercase and lowercase letters.

Self Check

What is the value of `s.length()` if `s` is

a. the empty string `""`?
b. the string `" "` containing a space?
c. `null`?

Answer: (a) 0; (b) 1; (c) an exception occurs.

Self Check

Which of the following comparisons are syntactically incorrect? Which of them are syntactically correct, but logically questionable?

```
String a = "1";
String b = "one";
double x = 1;
double y = 3 * (1.0 / 3);
```

a. a == "1"
b. a == null
c. a.equals("")
d. a == b
e. a == x
f. x == y
g. x - y == null
h. x.equals(y)

Answer: Syntactically incorrect: e, g, h. Logically questionable: a, d, f.

Multiple Alternatives: Sequences of Comparisons

-
  ```
  if ($condition_1$)
     $statement_1$;
  else if ($condition_2$)
     $statement_2$;
     ...
  else
     $statement_4$;
  ```
- The first matching condition is executed
- Order matters:

```
if (richter >= 0) // always passes
   r = "Generally not felt by people";
else if (richter >= 3.5) // not tested
   r = "Felt by many people, no destruction";
...
```

Multiple Alternatives: Sequences of Comparisons

- Don’t omit `else`:

```
if (richter >= 8.0)
   r = "Most structures fall";
if (richter >= 7.0) // omitted else--ERROR
   r = "Many buildings destroyed";
```

Earthquake.java

```
/**
   A class that describes the effects of an earthquake.
*/
public class Earthquake
{
   private double richter;

   /**
      Constructs an Earthquake object.
      @param magnitude the magnitude on the Richter scale
   */
   public Earthquake(double magnitude)
   {
      richter = magnitude;
   }

```

Earthquake.java (cont.)

```
    /**
        Gets a description of the effect of the earthquake.
        @return the description of the effect
    */
    public String getDescription()
    {
        String r;
        if (richter >= 8.0)
            r = "Most structures fall";
        else if (richter >= 7.0)
            r = "Many buildings destroyed";
        else if (richter >= 6.0)
            r = "Many buildings considerably damaged, some collapse";
        else if (richter >= 4.5)
            r = "Damage to poorly constructed buildings";
        else if (richter >= 3.5)
            r = "Felt by many people, no destruction";
        else if (richter >= 0)
            r = "Generally not felt by people";
        else
            r = "Negative numbers are not valid";
        return r;
    }
}
```

EarthquakeRunner.java

```java
import java.util.Scanner;

/**
    This program prints a description of an earthquake of a given magnitude.
*/
public class EarthquakeRunner
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("Enter a magnitude on the Richter scale: ");
      double magnitude = in.nextDouble();
      Earthquake quake = new Earthquake(magnitude);
      System.out.println(quake.getDescription());
   }
}
```

Program Run:

```
Enter a magnitude on the Richter scale: 7.1
Many buildings destroyed
```

Multiple Alternatives: Nested Branches

- Branch inside another branch:

```
if ($condition_1$)
{
   if ($condition_{1a}$)
      $statement_{1a}$;
   else
      $statement_{1b}$;
}
else
   $statement_2$;
```

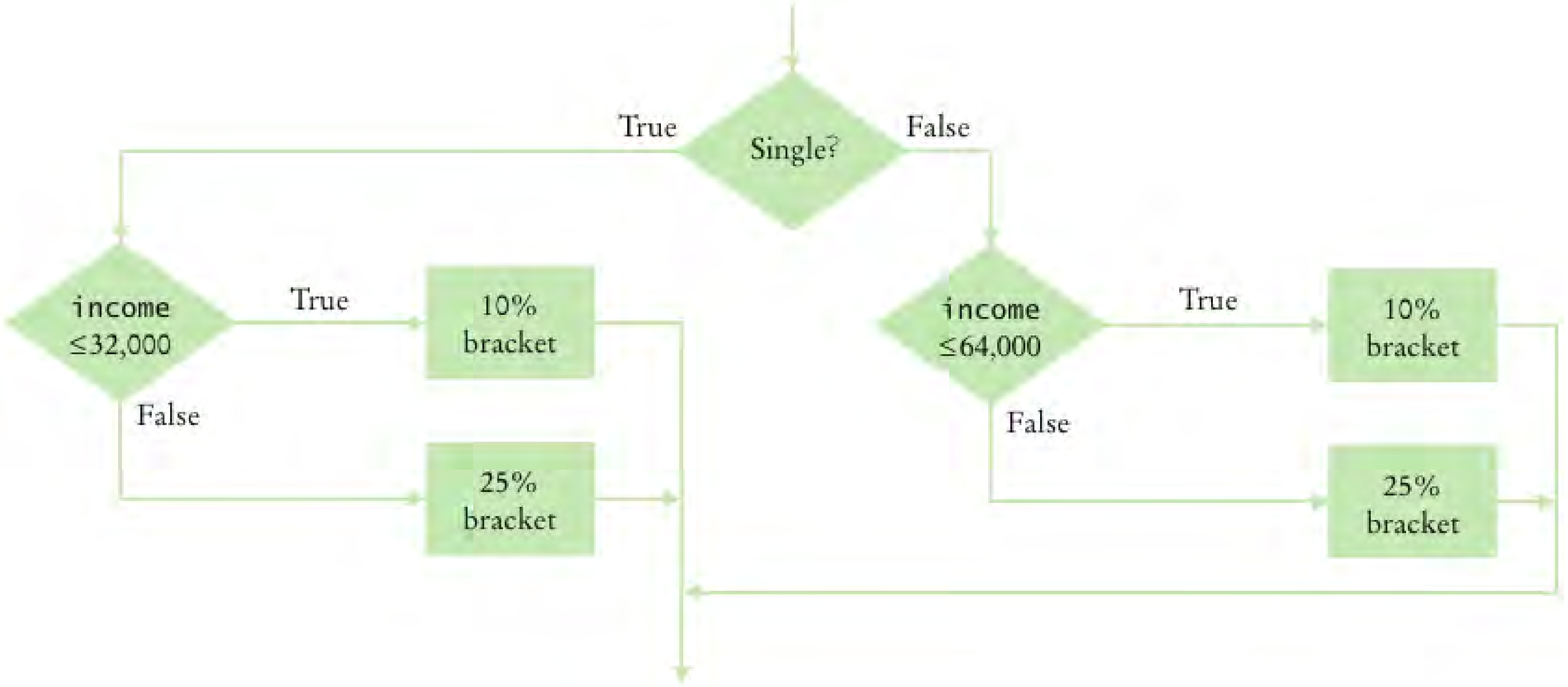
Tax Schedule

If your filing status is Single		If your filing status is Married	
Tax Bracket	**Percentage**	**Tax Bracket**	**Percentage**
$0 ... $32,000	10%	0 ... $64,000	10%
Amount over $32,000	25%	Amount over $64,000	25%

Nested Branches

- Compute taxes due, given filing status and income figure:
  1. *branch on the filing status*
  2. *for each filing status, branch on income level*
- The two-level decision process is reflected in two levels of `if` statements
- We say that the income test is *nested* inside the test for filing status

Nested Branches



TaxReturn.java

```
/**
   A tax return of a taxpayer in 2008.
*/
public class TaxReturn
{
   public static final int SINGLE = 1;
   public static final int MARRIED = 2;

   private static final double RATE1 = 0.10;
   private static final double RATE2 = 0.25;
   private static final double RATE1_SINGLE_LIMIT = 32000;
   private static final double RATE1_MARRIED_LIMIT = 64000;

   private double income;
   private int status;

```

TaxReturn.java (cont.)

```
   /**
      Constructs a TaxReturn object for a given income and
      marital status.
      @param anIncome the taxpayer income
      @param aStatus either SINGLE or MARRIED
   */
   public TaxReturn(double anIncome, int aStatus)
   {
      income = anIncome;
      status = aStatus;
   }

   public double getTax()
   {
      double tax1 = 0;
      double tax2 = 0;

```

TaxReturn.java (cont.)

```
      if (status == SINGLE)
      {
         if (income <= RATE1_SINGLE_LIMIT)
         {
            tax1 = RATE1 * income;
         }
         else
         {
            tax1 = RATE1 * RATE1_SINGLE_LIMIT;
            tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
         }
      }
      else
      {
         if (income <= RATE1_MARRIED_LIMIT)
         {
            tax1 = RATE1 * income;
         }
         else
         {
            tax1 = RATE1 * RATE1_MARRIED_LIMIT;
            tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
         }
      }

      return tax1 + tax2;
   }
}
```

TaxCalculator.java

```java
import java.util.Scanner;

/**
    This program calculates a simple tax return.
*/
public class TaxCalculator
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Please enter your income: ");
        double income = in.nextDouble();

        System.out.print("Are you married? (Y/N) ");
        String input = in.next();
        int status;
        if (input.equalsIgnoreCase("Y"))
           status = TaxReturn.MARRIED;
        else
           status = TaxReturn.SINGLE;
        TaxReturn aTaxReturn = new TaxReturn(income, status);

        System.out.println("Tax: "
              + aTaxReturn.getTax());
    }
}
```

TaxCalculator.java (cont.)

Program Run:

```
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
```

Self Check

The `if/else/else` statement for the earthquake strength first tested for higher values, then descended to lower values. Can you reverse that order?

Answer: Yes, if you also reverse the comparisons:

```
if (richter < 3.5)
   r = "Generally not felt by people";
else if (richter < 4.5)
   r =  "Felt by many people, no destruction";
else if (richter < 6.0)
   r =  "Damage to poorly constructed buildings";
...
```

Self Check

Some people object to higher tax rates for higher incomes, claiming that you might end up with less money after taxes when you get a raise for working hard. What is the flaw in this argument?

Answer: The higher tax rate is only applied on the income in the higher bracket. Suppose you are single and make $31,900. Should you try to get a $200 raise? Absolutely: you get to keep 90 percent of the first $100 and 75 percent of the next $100.

Using Boolean Expressions: The `boolean` Type

- George Boole (1815-1864): pioneer in the study of logic value of expression `amount < 1000` is `true` or `false`
- `boolean` type: one of these 2 truth values

Using Boolean Expressions: Predicate Method

- A predicate method returns a `boolean` value:

```
public boolean isOverdrawn()
{
   return balance < 0;
}
```

- Use in conditions:

```
if (harrysChecking.isOverdrawn())
```

- Useful predicate methods in `Character` class:

```
isDigit
isLetter
isUpperCase
isLowerCase
```

Using Boolean Expressions: Predicate Method

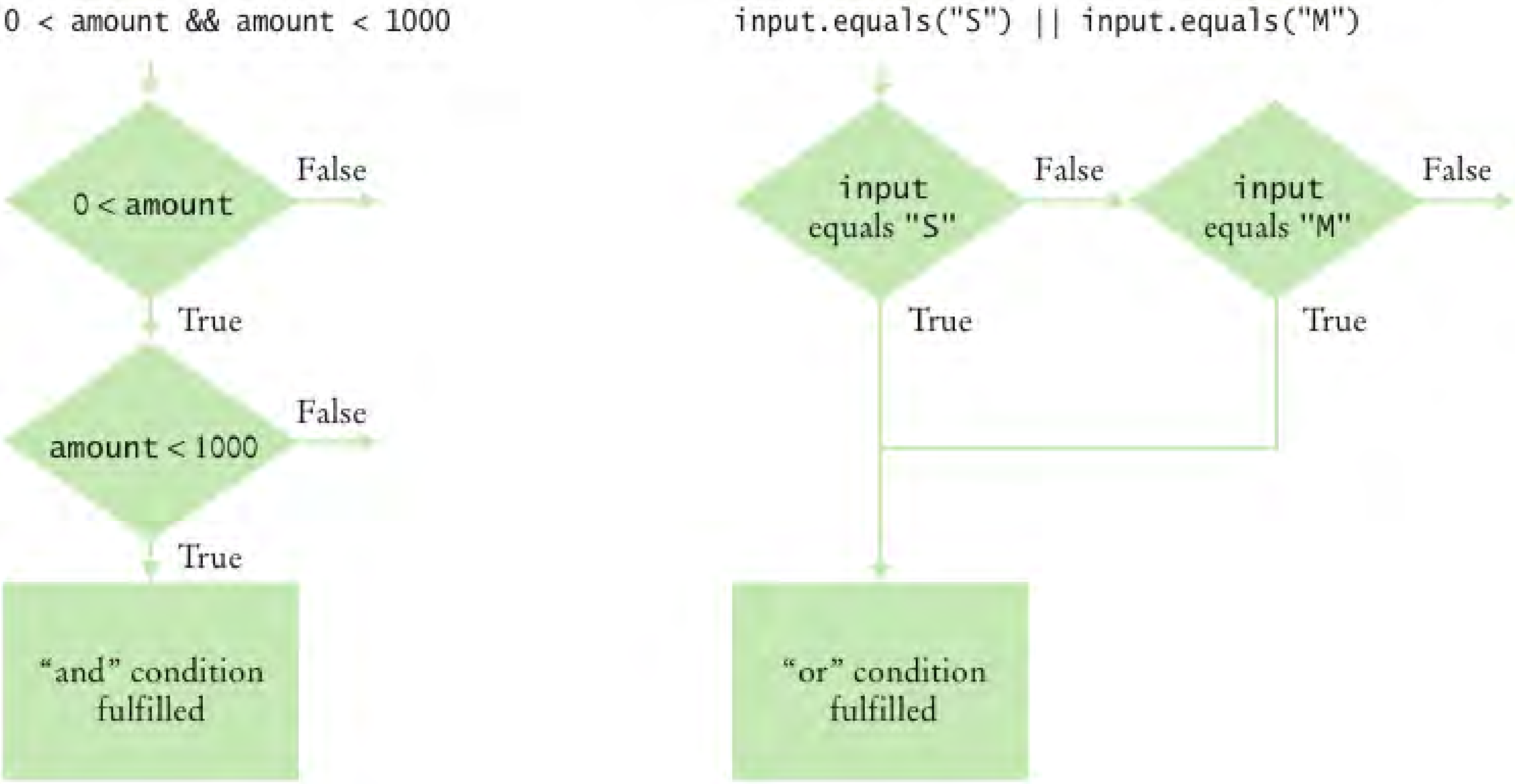
- `if (Character.isUpperCase(ch)) ...`
- Useful predicate methods in `Scanner` class: `hasNextInt()` and `hasNextDouble()`:

```
if (in.hasNextInt()) n = in.nextInt();
```

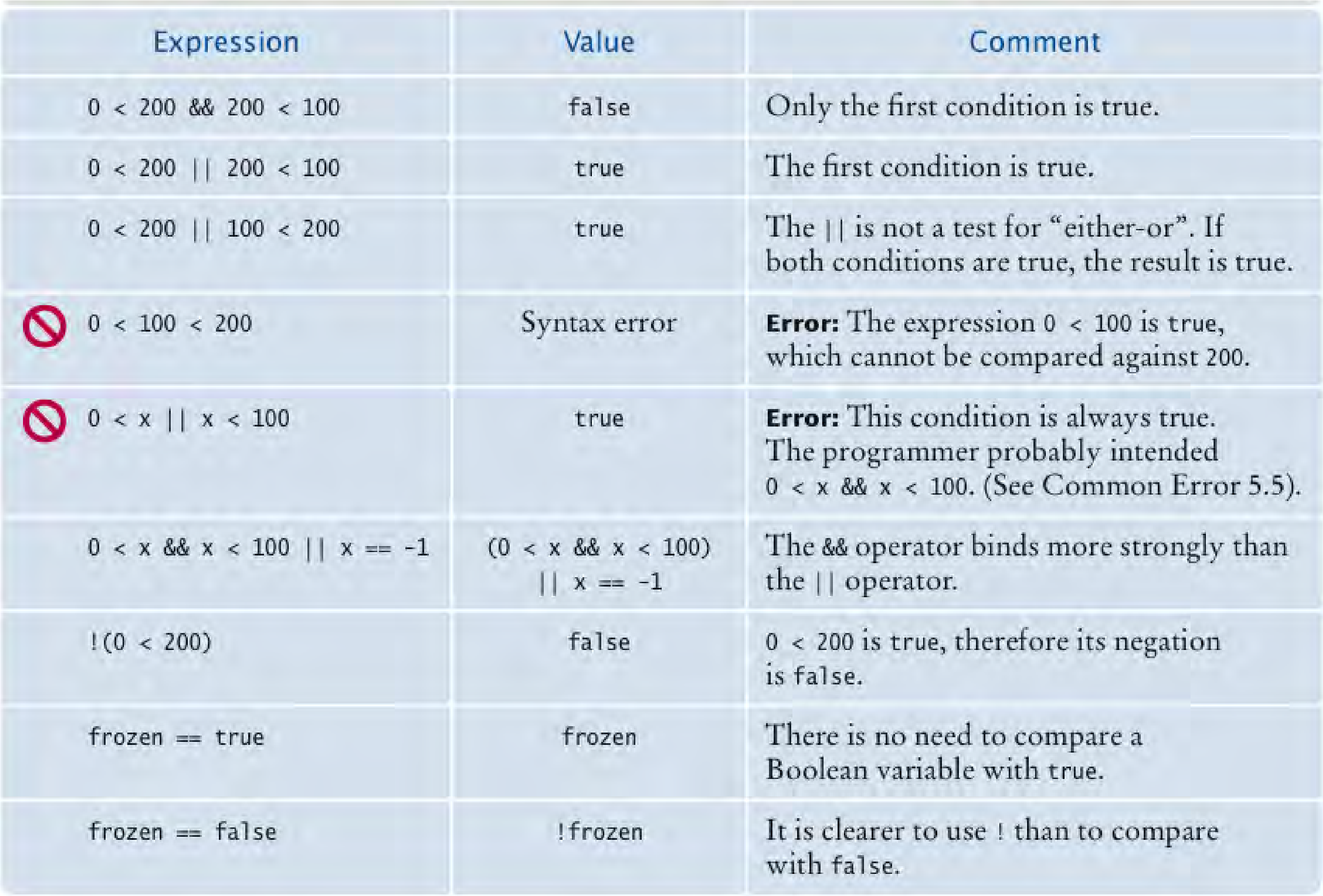
Using Boolean Expressions: The Boolean Operators

- `&&` and
- `||` or
- `!` not
- `if (0 < amount && amount < 1000) ...`
- `if (input.equals("S") || input.equals("M")) ...`
- `if (!input.equals("S")) ...`

`&&` and `||` Operators



Boolean Operators



Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for “either-or”. If both conditions are true, the result is true.
🚫 `0 < 100 < 200`	Syntax error	**Error:** The expression `0 < 100` is `true`, which cannot be compared against `200`.
🚫 `0 < x || x < 100`	`true`	**Error:** This condition is always true. The programmer probably intended `0 < x && x < 100`. (See Common Error 5.5).

`0 < x && x < 100 || x == -1`	`(0 < x && x < 100)`
	`|| x == -1`
		The `&&` operator binds more strongly than the `||` operator.

`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Truth Tables

A	B	A && B
true	true	true
true	false	false
false	*Any*	false

A	B	A || B
true	*Any*	true
false	true	true
false	false	false

A	!A
true	false
false	true

Using Boolean Variables

- `private boolean married;`
- Set to truth value:

```
married = input.equals("M");
```

- Use in conditions:

```
if (married) ... else ...
if (!married) ...
```

- Also called *flag*
- It is considered gauche to write a test such as

```
if (married == true) ... // Don't
```

- Just use the simpler test

```
if (married) ...
```

Self Check

When does the statement

```
system.out.println (x > 0 || x < 0);
```

print `false`?

Answer: When `x` is zero.

Self Check

Rewrite the following expression, avoiding the comparison with `false`:

```
if (character.isDigit(ch) == false) ...
```

Answer: `if (!Character.isDigit(ch)) ...`

Code Coverage

- **Black-box testing:** Test functionality without consideration of internal structure of implementation
- **White-box testing:** Take internal structure into account when designing tests
- **Test coverage:** Measure of how many parts of a program have been tested
- Make sure that each part of your program is exercised at least once by one test case

  E.g., make sure to execute each branch in at least one test case
- Include boundary test cases: Legal values that lie at the boundary of the set of acceptable inputs
- Tip: Write first test cases before program is written completely → gives insight into what program should do

Self Check

How many test cases do you need to cover all branches of the `getDescription` method of the `Earthquake` class?

Answer: 7.

Self Check

Give a boundary test case for the `EarthquakeRunner` program. What output do you expect?

Answer: An input of 0 should yield an output of `"Generally not felt by people"`. (If the output is `"Negative numbers are not allowed"`, there is an error in the program.)

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

AGENDA

- Objects (cont.)
- Classes
- Decisions
- Iteration

BASIC CONCEPTS OF JAVA 2

ITERATION

Chapter Goals

- To be able to program loops with the `while` and `for` statements
- To avoid infinite loops and off-by-one errors
- To be able to use common loop algorithms
- To understand nested loops
- To implement simulations
- To learn about the debugger

`while` Loops

- A `while` statement executes a block of code repeatedly
- A condition controls how often the loop is executed

```
while (condition)
    statement
```

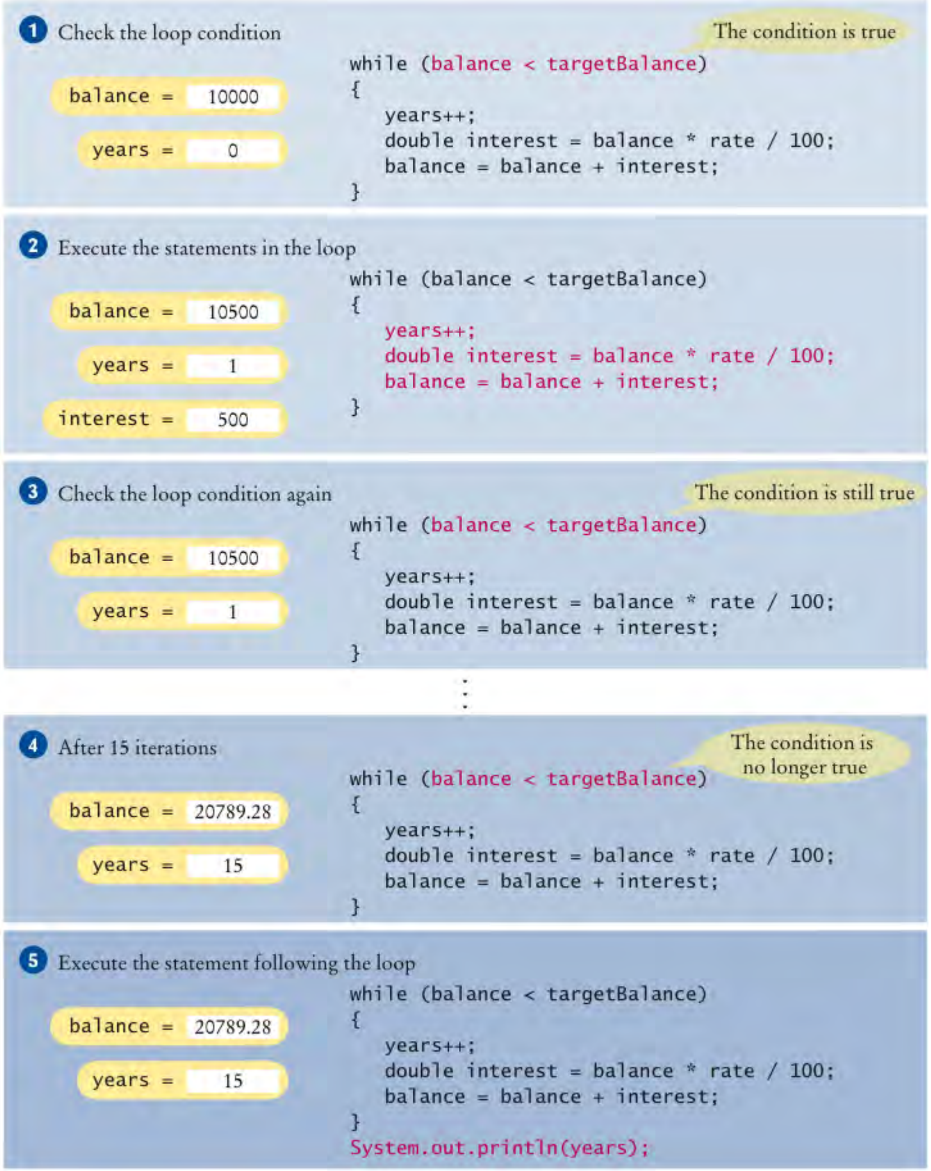
- Most commonly, the statement is a block statement (set of statements delimited by `{ }`)

Calculating the Growth of an Investment

- Want to know when has the bank account reached a particular balance:

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Execution of a `while` Loop



1 Check the loop condition

The condition is true

balance = 10000

years = 0

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

2 Execute the statements in the loop

balance = 10500

years = 1

interest = 500

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

3 Check the loop condition again

The condition is still true

balance = 10500

years = 1

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

⋮

4 After 15 iterations

The condition is no longer true

balance = 20789.28

years = 15

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

5 Execute the statement following the loop

balance = 20789.28

years = 15

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
System.out.println(years);
```

Investment.java

```java
/**
   A class to monitor the growth of an investment that
   accumulates interest at a fixed annual rate.
*/
public class Investment
{
   private double balance;
   private double rate;
   private int years;

   /**
      Constructs an Investment object from a starting balance and
      interest rate.
      @param aBalance the starting balance
      @param aRate the interest rate in percent
   */
   public Investment(double aBalance, double aRate)
   {
      balance = aBalance;
      rate = aRate;
      years = 0;
   }

```

Investment.java (cont.)

```
   /**
      Keeps accumulating interest until a target balance has
      been reached.
      @param targetBalance the desired balance
   */
   public void waitForBalance(double targetBalance)
   {
      while (balance < targetBalance)
      {
         years++;
         double interest = balance * rate / 100;
         balance = balance + interest;
      }
   }

   /**
      Gets the current investment balance.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }
```

Investment.java (cont.)

```
   /**
      Gets the number of years this investment has accumulated
      interest.
      @return the number of years since the start of the investment
   */
   public int getYears()
   {
      return years;
   }
}
```

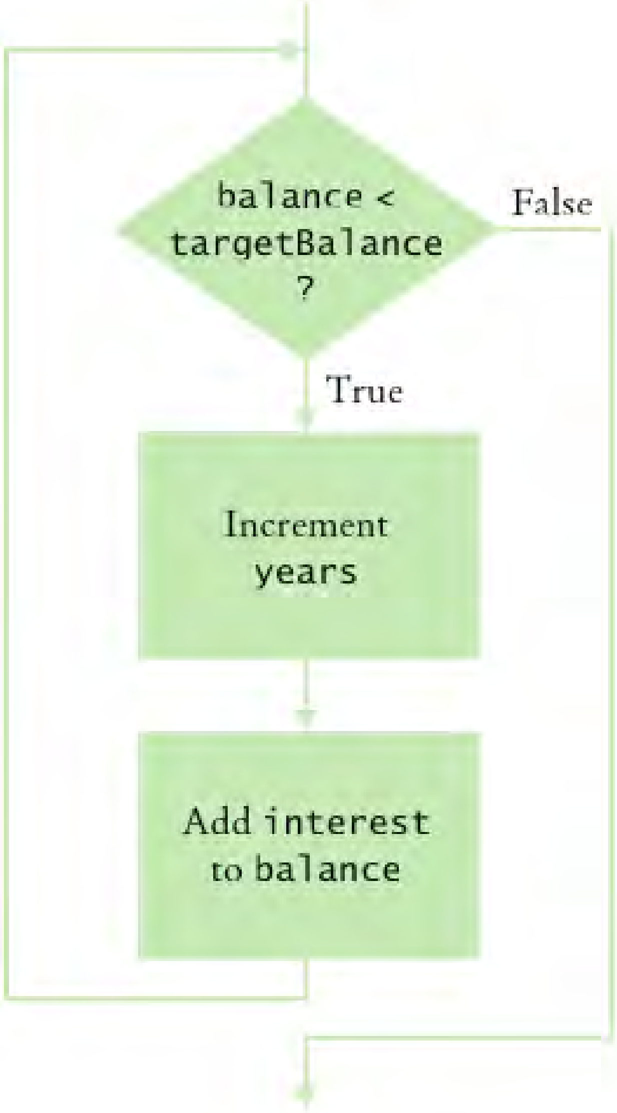
InvestmentRunner.java

```java
/**
    This program computes how long it takes for an investment
    to double.
*/
public class InvestmentRunner
{
   public static void main(String[] args)
   {
      final double INITIAL_BALANCE = 10000;
      final double RATE = 5;
      Investment invest = new Investment(INITIAL_BALANCE, RATE);
      invest.waitForBalance(2 * INITIAL_BALANCE);
      int years = invest.getYears();
      System.out.println("The investment doubled after "
            + years + " years");
   }
}
```

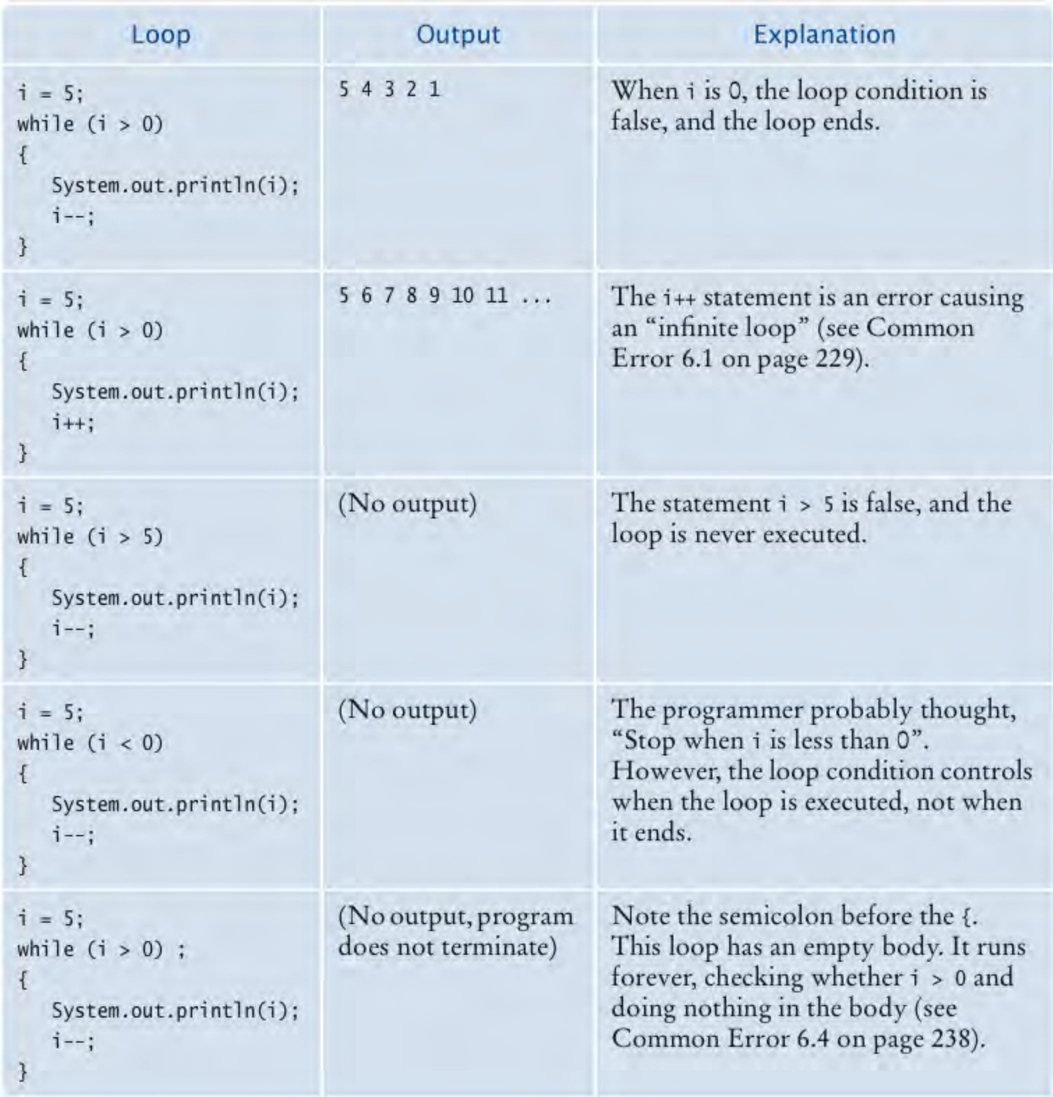
Program Run:

```
The investment doubled after 15 years
```

`while` Loop Flowchart



while Loop Examples



Loop	Output	Explanation

i = 5;
while (i > 0)
{
System.out.println(i);
i--;
}
	5 4 3 2 1	When i is 0, the loop condition is false, and the loop ends.

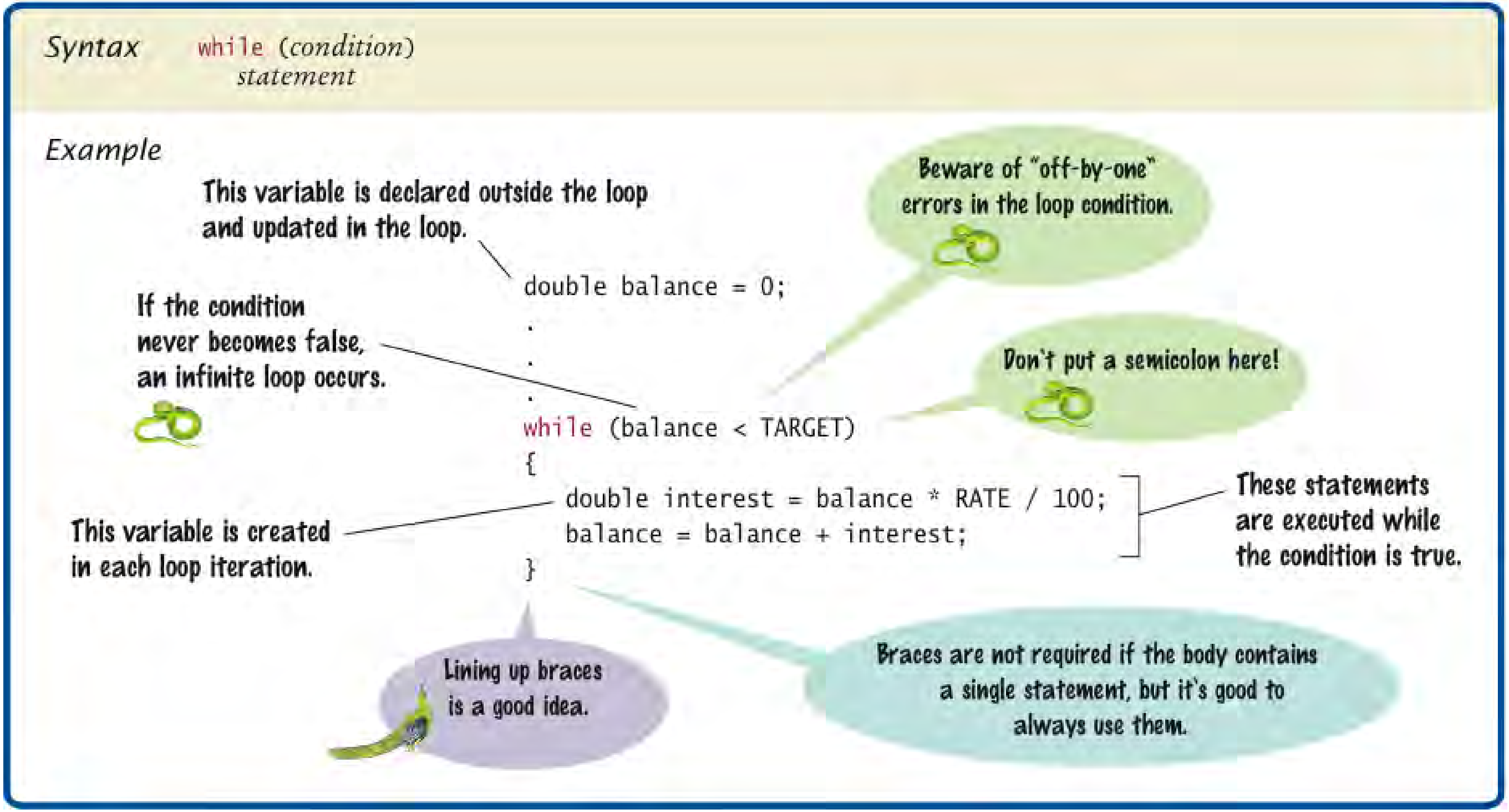
i = 5;
while (i > 0)
{
System.out.println(i);
i++;
}
	5 6 7 8 9 10 11 ...	The i++ statement is an error causing an "infinite loop" (see Common Error 6.1 on page 229).

i = 5;
while (i > 5)
{
System.out.println(i);
i--;
}
	(No output)	The statement i > 5 is false, and the loop is never executed.

i = 5;
while (i < 0)
{
System.out.println(i);
i--;
}
	(No output)	The programmer probably thought, "Stop when i is less than 0". However, the loop condition controls when the loop is executed, not when it ends.

i = 5;
while (i > 0) ;
{
System.out.println(i);
i--;
}
	(No output, program does not terminate)	Note the semicolon before the {. This loop has an empty body. It runs forever, checking whether i > 0 and doing nothing in the body (see Common Error 6.4 on page 238).

Syntax The `while` Statement



Syntax

```
while (condition)
    statement
```

Example

```
double balance = 0;
.
.
.
while (balance < TARGET)
{
   double interest = balance * RATE / 100;
   balance = balance + interest;
}
```

This variable is declared outside the loop and updated in the loop.

If the condition never becomes false, an infinite loop occurs.

This variable is created in each loop iteration.

Beware of "off-by-one" errors in the loop condition.

Don't put a semicolon here!

These statements are executed while the condition is true.

Lining up braces is a good idea.

Braces are not required if the body contains a single statement, but it's good to always use them.

Self Check

How often is the following statement in the loop executed?

```
   while (false) statement;
```

Self Check

What would happen if `RATE` was set to `0` in the `main` method of the `InvestmentRunner` program?

Answer: The `waitForBalance` method would never return due to an infinite loop.

Common Error: Infinite Loops

- Example:

```
int years = 0;
while (years < 20)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

- Loop runs forever - must kill program

Common Error: Infinite Loops

- Example:

```
int years = 20;
while (years > 0)
{
   years++; // Oops, should have been years--
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

- Loop runs forever - must kill program

Common Error: Off-by-One Errors

- **Off-by-one error:** a loop executes one too few, or one too many, times
- Example:

```
int years = 0;
while (balance < 2 * initialBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
System.out.println("The investment reached the target after
" + years + " years.");
```

- Should `years` start at 0 or 1?
- Should the test be `<` or `<=`?

Avoiding Off-by-One Error

- Look at a scenario with simple values:
  initial `balance: $100`
  interest `rate: 50%`
  after year 1, the `balance` is $150
  after year 2 it is $225, or over $200
  so the investment doubled after 2 years
  the loop executed two times, incrementing `years` each time
  Therefore: `years` must start at 0, not at 1.
- interest `rate`: `100%`
  after one year: `balance` is `2 * initialBalance`
  loop should stop
  Therefore: must use <
- Think, don't compile and try at random

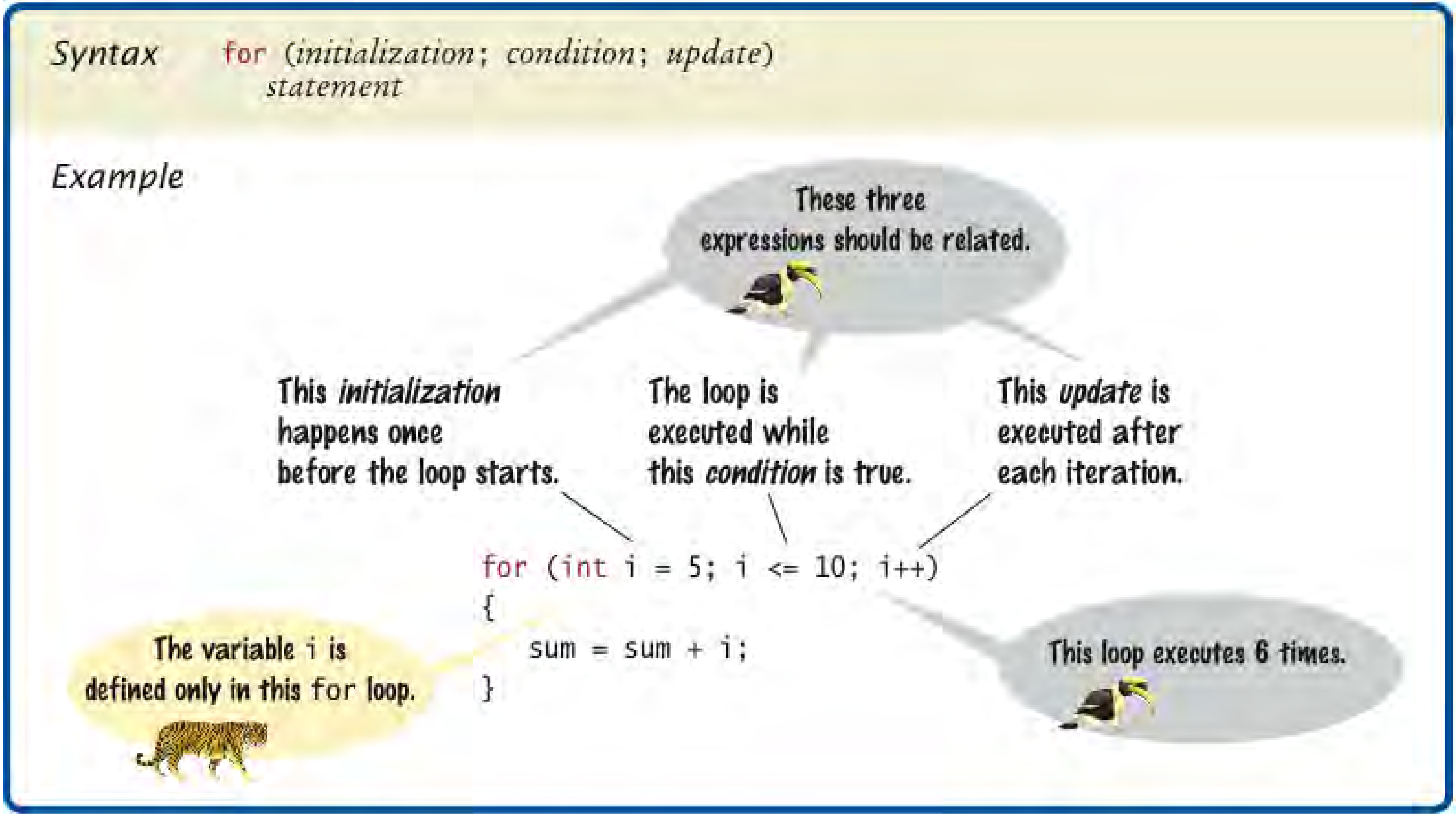
for Loops

- Example:

```
for (int i = 1; i <= n; i++)
  {
     double interest = balance * rate / 100;
     balance = balance + interest;
  }
```

- Use a for loop when a variable runs from a starting value to an ending value with a constant increment or decrement

Syntax The `for` Statement



Syntax for (*initialization*; *condition*; *update*)
statement

Example

These three expressions should be related.

This *initialization* happens once before the loop starts.

The loop is executed while this *condition* is true.

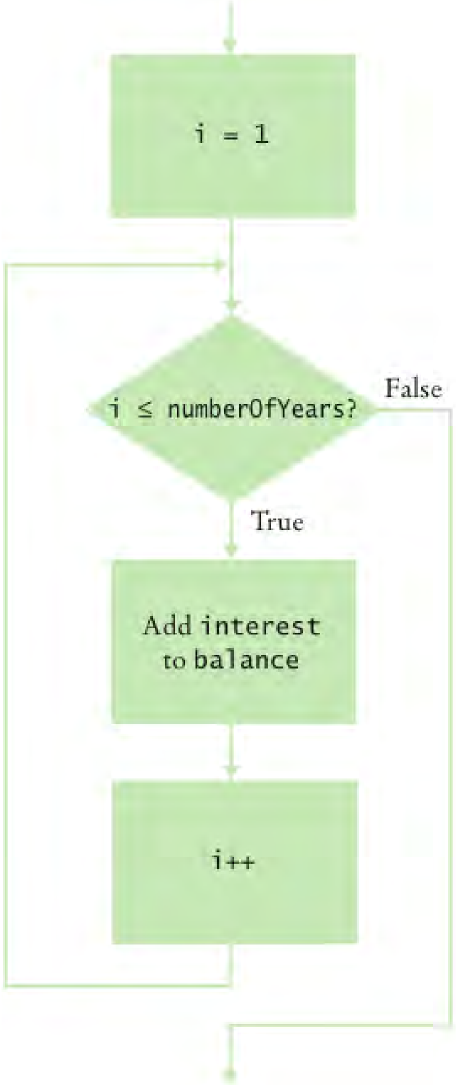
This *update* is executed after each iteration.

```
for (int i = 5; i <= 10; i++)
{
   sum = sum + i;
}
```

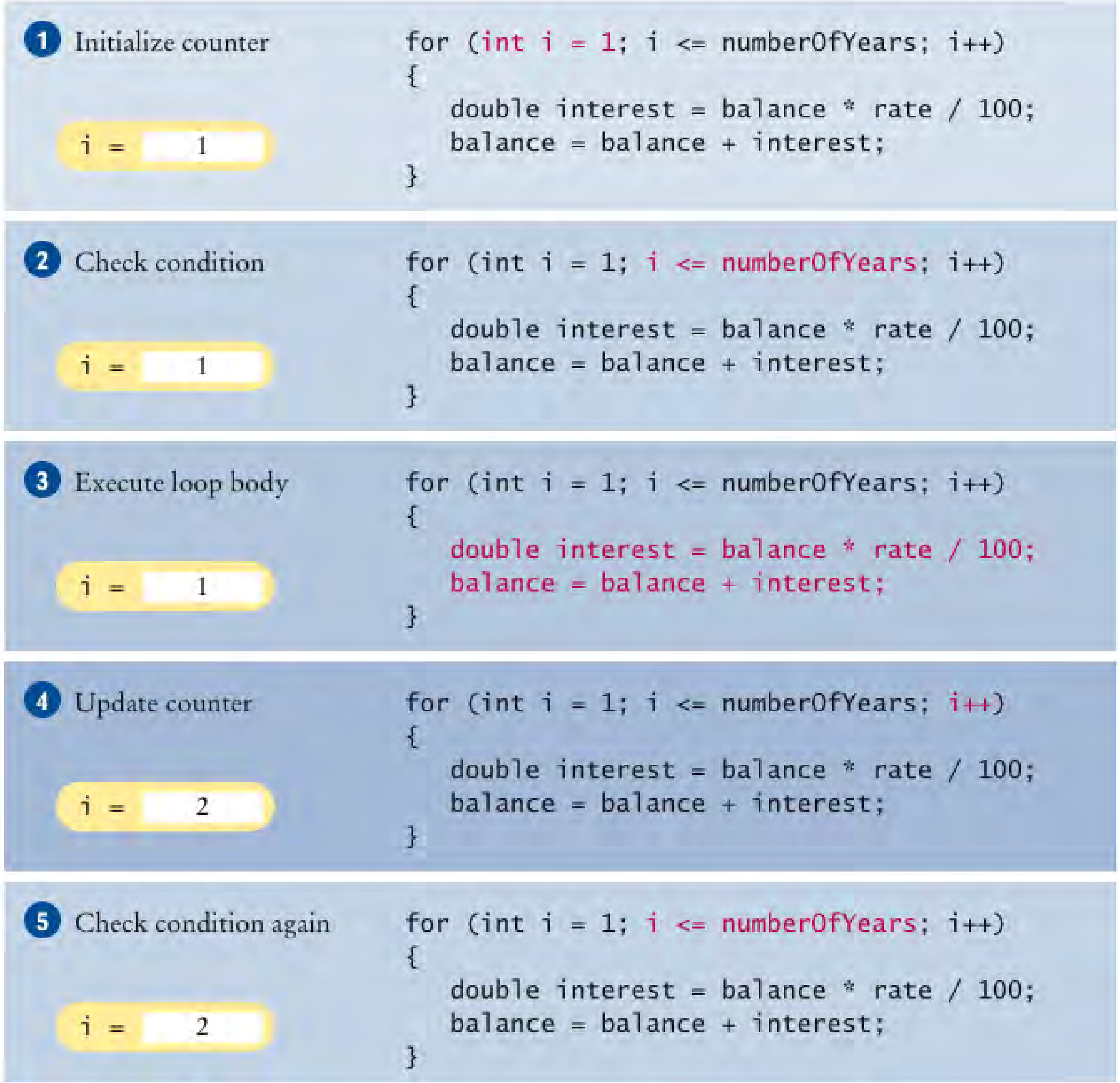
The variable i is defined only in this for loop.

This loop executes 6 times.

for Loop Flowchart



Execution of a `for` Loop



1 Initialize counter

i = 1

```
for (int i = 1; i <= numberOfYears; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

2 Check condition

i = 1

```
for (int i = 1; i <= numberOfYears; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

3 Execute loop body

i = 1

```
for (int i = 1; i <= numberOfYears; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

4 Update counter

i = 2

```
for (int i = 1; i <= numberOfYears; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

5 Check condition again

i = 2

```
for (int i = 1; i <= numberOfYears; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Investment.java

```
/**
   A class to monitor the growth of an investment that
   accumulates interest at a fixed annual rate
*/
public class Investment
{
   private double balance;
   private double rate;
   private int years;

   /**
      Constructs an Investment object from a starting balance and
      interest rate.
      @param aBalance the starting balance
      @param aRate the interest rate in percent
   */
   public Investment(double aBalance, double aRate)
   {
      balance = aBalance;
      rate = aRate;
      years = 0;
   }

```

Investment.java (cont.)

```java
   /**
      Keeps accumulating interest until a target balance has
      been reached.
      @param targetBalance the desired balance
   */
   public void waitForBalance(double targetBalance)
   {
      while (balance < targetBalance)
      {
         years++;
         double interest = balance * rate / 100;
         balance = balance + interest;
      }
   }

   /**
      Keeps accumulating interest for a given number of years.
      @param numberOfYears the number of years to wait
   */
   public void waitYears(int numberOfYears)
   {
      for (int i = 1; i <= numberOfYears; i++)
      {
         double interest = balance * rate / 100;
         balance = balance + interest;
      }
      years = years + n;
   }
```

Investment.java (cont.)

```java

   /**
      Gets the current investment balance.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }

   /**
      Gets the number of years this investment has accumulated
      interest.
      @return the number of years since the start of the investment
   */
   public int getYears()
   {
      return years;
   }
}
```

InvestmentRunner.java

```
/**
   This program computes how much an investment grows in
   a given number of years.
*/
public class InvestmentRunner
{
   public static void main(String[] args)
   {
      final double INITIAL_BALANCE = 10000;
      final double RATE = 5;
      final int YEARS = 20;
      Investment invest = new Investment(INITIAL_BALANCE, RATE);
      invest.waitYears(YEARS);
      double balance = invest.getBalance();
      System.out.printf("The balance after %d years is %.2f\n",
            YEARS, balance);
   }
}
```

Program Run:

```
The balance after 20 years is 26532.98
```

Self Check

Rewrite the `for` loop in the `waitYears` method as a `while` loop.

Answer:

```
int i = 1;
while (i <= n)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
   i++;
}
```

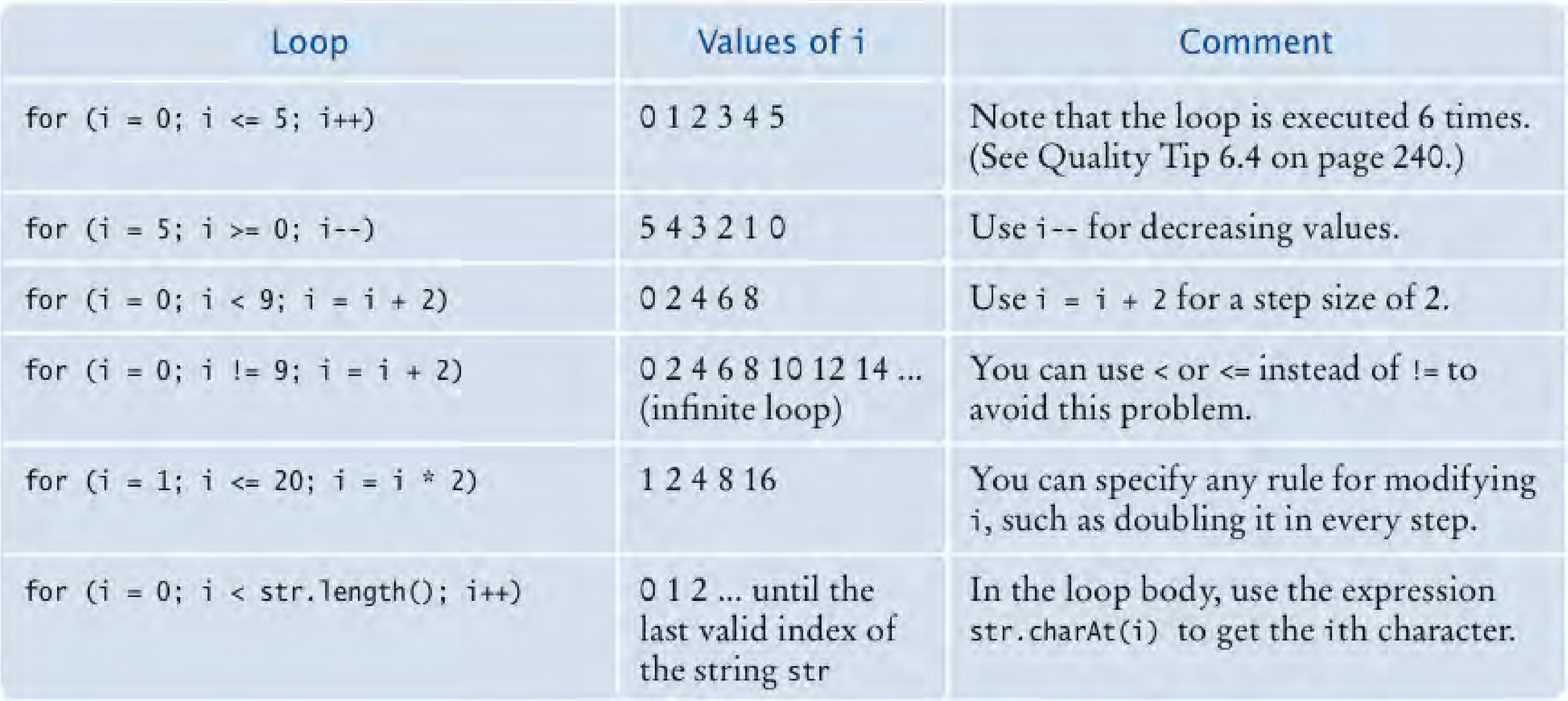
Self Check

How many times does the following for loop execute?

```
   for (i = 0; i <= 10; i++)
      System.out.println(i * i);
```

Answer: 11 times.

`for` Loop Examples



Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Quality Tip 6.4 on page 240.)
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i = i + 2)`	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use `<` or `<=` instead of `!=` to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying `i`, such as doubling it in every step.
`for (i = 0; i < str.length(); i++)`	0 1 2 ... until the last valid index of the string str	In the loop body, use the expression `str.charAt(i)` to get the ith character.

Common Errors: Semicolons

- A missing semicolon:

```
for (years = 1;
    (balance = balance + balance * rate / 100) < targetBalance;
    years++)
        System.out.println(years);
```

- A semicolon that shouldn’t be there:

```
sum = 0;
for (i = 1; i <= 10; i++);
   sum = sum + i;
System.out.println(sum);
```

Common Loop Algorithm: Computing a Total

- Example - keep a *running total*: a variable to which you add each input value:

```
double total = 0;
while (in.hasNextDouble())
{
   double input = in.nextDouble();
   total = total + input;
}
```

Common Loop Algorithm: Counting Matches

- Example - count how many uppercase letters are in a string:

```
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++)
{
   char ch = str.charAt(i);
   if (Character.isUpperCase(ch))
   {
      upperCaseLetters++;
   }
}
```

Common Loop Algorithm: Finding the First Match

- Example - find the first lowercase letter in a string:

```
boolean found = false;
char ch = '?';
int position = 0;
while (!found && position < str.length())
{
   ch = str.charAt(position);
   if (Character.isLowerCase(ch)) { found = true; }
   else { position++; }
}
```

Common Loop Algorithm: Prompting Until a Match is Found

- Example - Keep asking the user to enter a positive value < 100 until the user provides a correct input:

```
boolean valid = false;
double input;
while (!valid)
{
   System.out.print("Please enter a positive value < 100: ");
   input = in.nextDouble();
   if (0 < input && input < 100) { valid = true; }
   else { System.out.println("Invalid input."); }
}
```

Common Loop Algorithm: Comparing Adjacent Values

- Example - check whether a sequence of inputs contains adjacent duplicates such as `1 7 2 9 9 4 9`:

```
double input = in.nextDouble();
while (in.hasNextDouble())
{
   double previous = input;
   input = in.nextDouble();
   if (input == previous) { System.out.println("Duplicate
  input"); }
}
```

Common Loop Algorithm: Processing Input with Sentinel Values

- Example - process a set of values
- **Sentinel value:** Can be used for indicating the end of a data set
- `0` or `–1` make poor sentinels; better to use `Q`:

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
   We are done
else
{
   double x = Double.parseDouble(input);
   . . .
}
```

Loop and a Half

- Sometimes termination condition of a loop can only be evaluated in the middle of the loop
- Then, introduce a boolean variable to control the loop:

```
boolean done = false;
while (!done)
{
   Print prompt
   String input = read input;
   if (end of input indicated)
      done = true;
   else
   {
      Process input
   }
}
```

DataAnalyzer.java

```
import java.util.Scanner;

/**
   This program computes the average and maximum of a set
   of input values.
*/
public class DataAnalyzer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      DataSet data = new DataSet();

      boolean done = false;
      while (!done)
      {
         System.out.print("Enter value, Q to quit: ");
         String input = in.next();
         if (input.equalsIgnoreCase("Q"))
            done = true;
         else
         {
            double x = Double.parseDouble(input);
            data.add(x);
         }
      }

      System.out.println("Average = " + data.getAverage());
      System.out.println("Maximum = " + data.getMaximum());
   }
}
```

DataSet.java

```
/**
    Computes information about a set of data values.
*/
public class DataSet
{
   private double sum;
   private double maximum;
   private int count;

   /**
      Constructs an empty data set.
   */
   public DataSet()
   {
      sum = 0;
      count = 0;
      maximum = 0;
   }

   /**
      Adds a data value to the data set
      @param x a data value
   */
```

DataSet.java (cont.)

```
   public void add(double x)
   {
      sum = sum + x;
      if (count == 0 || maximum < x) maximum = x;
      count++;
   }

   /**
      Gets the average of the added data.
      @return the average or 0 if no data has been added
   */
   public double getAverage()
   {
      if (count == 0) return 0;
      else return sum / count;
   }

   /**
      Gets the largest of the added data.
      @return the maximum or 0 if no data has been added
   */
   public double getMaximum()
   {
      return maximum;
   }
}
```

DataSet.java (cont.)

Program Run:

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

Self Check

How do you compute the total of all positive inputs?

Answer:

```
double total = 0;
while (in.hasNextDouble())
{
   double input = in.nextDouble();
   if (value > 0) total = total + input;
}
```

Self Check

What happens with the algorithm in Comparing Adjacent Values, when no input is provided at all? How can you overcome that problem?

Answer: The initial call to `in.nextDouble()` fails, terminating the program. One solution is to do all input in the loop and introduce a Boolean variable that checks whether the loop is entered for the first time.

```
double input = 0;
boolean first = true;
while (in.hasNextDouble())
{
   double previous = input;
   input = nextDouble();
   if (first) { first = false; }
   else if (input == previous) { System.out.println("Duplicate
input"); }
}
```

Self Check

Why does the `DataAnalyzer` class call `in.next` and not `in.nextDouble`?

Answer: Because we don't know whether the next input is a number or the letter.

Self Check

Would the `DataSet` class still compute the correct maximum if you simplified the update of the `maximum` field in the add method to the following statement?

```
   if (maximum < x) maximum = x;
```

Answer: No. If *all* input values are negative, the maximum is also negative. However, the `maximum` field is initialized with 0. With this simplification, the maximum would be falsely computed as 0.

Nested Loops

- Create triangle shape:

```
[]
[][]
[][][]
[][][][]
```

- Loop through rows:

```
for (int i = 1; i <= n; i++)
{
   // make triangle row
}
```

- *Make triangle row* is another loop:

```
for (int j = 1; j <= i; j++)
   r = r + "[]";
r = r + "\n";
```

- Put loops together → Nested loops

Triangle.java

```java
/**
   This class describes triangle objects that can be displayed
   as shapes like this:
   []
   [][]
   [][][]
*/
public class Triangle
{
   private int width;

   /**
      Constructs a triangle.
      @param aWidth the number of [] in the last row of the triangle.
   */
   public Triangle(int aWidth)
   {
      width = aWidth;
   }

```

Triangle.java (cont.)

```
   /**
      Computes a string representing the triangle.
      @return a string consisting of [] and newline characters
   */
   public String toString()
   {
      String r = "";
      for (int i = 1; i <= width; i++)
      {
         // Make triangle row
         for (int j = 1; j <= i; j++)
            r = r + "[]";
         r = r + "\n";
      }
      return r;
   }
}
```

TriangleRunner.java

```java
/**
     This program prints two triangles.
*/
public class TriangleRunner
{
   public static void main(String[] args)
   {
      Triangle small = new Triangle(3);
      System.out.println(small.toString());

      Triangle large = new Triangle(15);
      System.out.println(large.toString());
   }
}
```

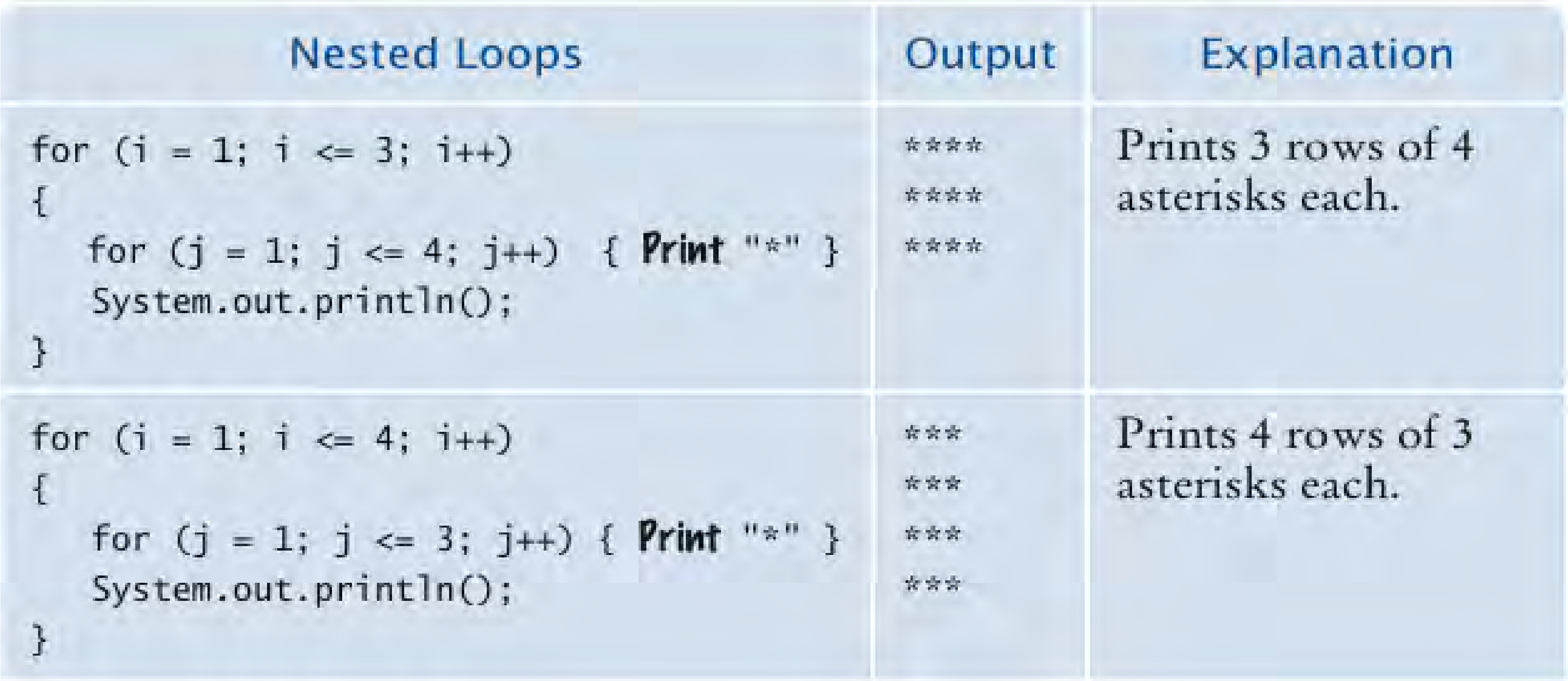
TriangleRunner.java (cont.)

Program Run:

```
[]
[] []
[] [] []

[]
[] []
[] [] []
[] [] [] []
[] [] [] [] []
[] [] [] [] [] []
[] [] [] [] [] [] []
[] [] [] [] [] [] [] []
[] [] [] [] [] [] [] [] []  [] [] [] [] [] [] [] [] [] []
[] [] [] [] [] [] [] [] [] [] []  [] [] [] [] [] [] [] [] [] [] [] []
[] [] [] [] [] [] [] [] [] [] [] [] []
[] [] [] [] [] [] [] [] [] [] [] [] [] []
[] [] [] [] [] [] [] [] [] [] [] [] [] [] []
```

Nested Loop Examples

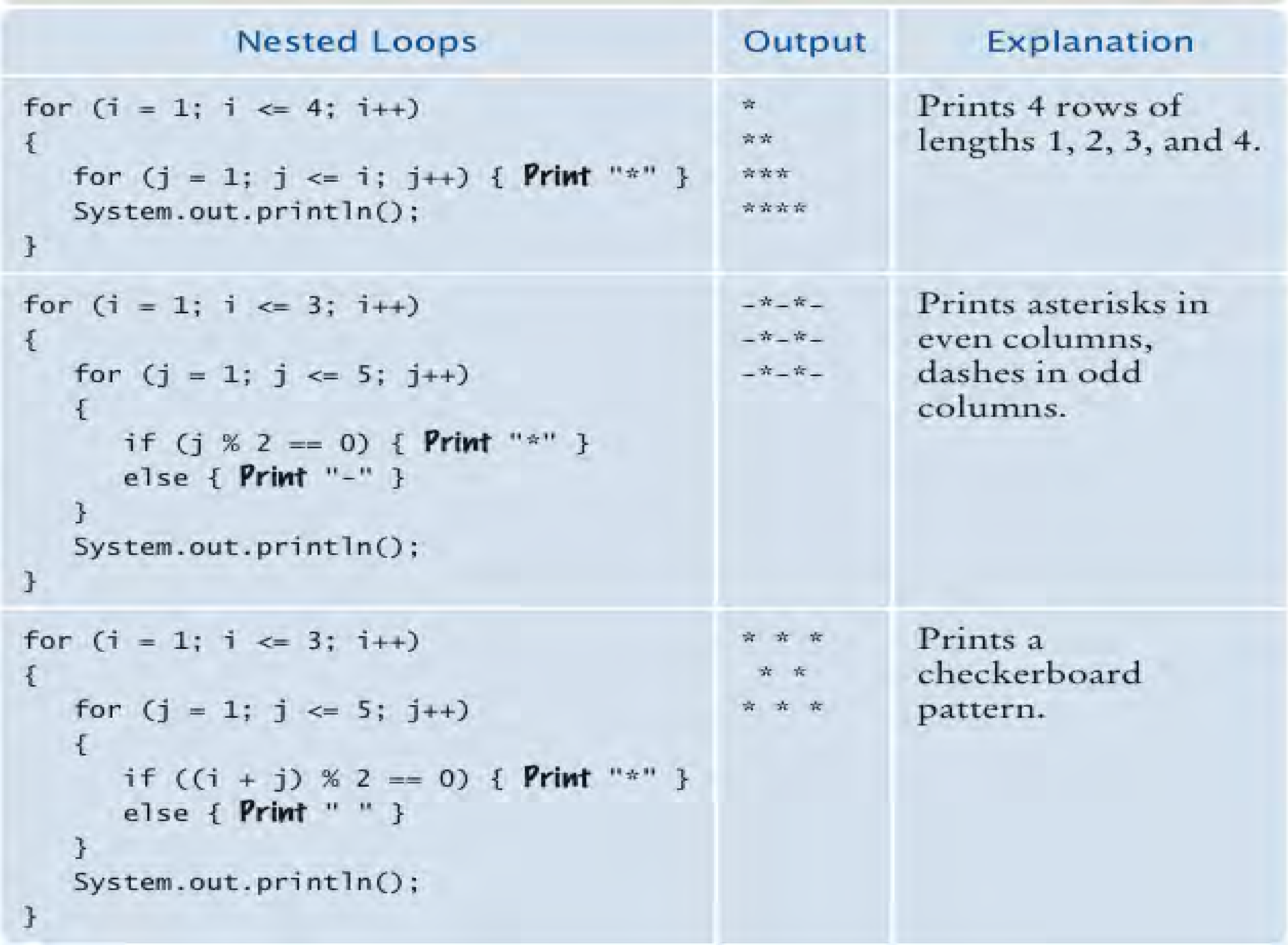


Nested Loops	Output	Explanation

for (i = 1; i <= 3; i++)
{
    for (j = 1; j <= 4; j++) { Print "*" }
    System.out.println();
}
	****
	****
	****
		Prints 3 rows of 4 asterisks each.

for (i = 1; i <= 4; i++)
{
    for (j = 1; j <= 3; j++) { Print "*" }
    System.out.println();
}
	***
	***
	***
	***
		Prints 4 rows of 3 asterisks each.

Nested Loop Examples



Nested Loops	Output	Explanation

`for (i = 1; i <= 4; i++)`
`{`
`   for (j = 1; j <= i; j++) { Print "*" }`
`   System.out.println();`
`}`
	`*`
	`**`
	`***`
	`****`
		Prints 4 rows of lengths 1, 2, 3, and 4.

`for (i = 1; i <= 3; i++)`
`{`
`   for (j = 1; j <= 5; j++)`
`   {`
`      if (j % 2 == 0) { Print "*" }`
`      else { Print "-" }`
`   }`
`   System.out.println();`
`}`
	`-*-*-`
	`-*-*-`
	`-*-*-`
		Prints asterisks in even columns, dashes in odd columns.

`for (i = 1; i <= 3; i++)`
`{`
`   for (j = 1; j <= 5; j++)`
`   {`
`      if ((i + j) % 2 == 0) { Print "*" }`
`      else { Print " " }`
`   }`
`   System.out.println();`
`}`
	`* * *`
	` * *`
	`* * *`
		Prints a checkerboard pattern.

Self Check

How would you modify the nested loops so that you print a square instead of a triangle?

Answer: Change the inner loop to `for (int j = 1; j <= width; j++)`

Self Check

What is the value of `n` after the following nested loops?

```
int n = 0;
for (int i = 1; i <= 5; i++)
   for (int j = 0; j < i; j++)
      n = n + j;
```

Answer: 20.

Random Numbers and Simulations

- In a simulation, you repeatedly generate random numbers and use them to simulate an activity
- Random number generator

```
Random generator = new Random();
int n = generator.nextInt(a); // 0 < = n < a
double x = generator.nextDouble(); // 0 <= x < 1
```

- Throw die (random number between 1 and 6)

```
int d = 1 + generator.nextInt(6);
```

Die.java

```java
import java.util.Random;

/**
   This class models a die that, when cast, lands on a random
   face.
*/
public class Die
{
   private Random generator;
   private int sides;

   /**
      Constructs a die with a given number of sides.
      @param s the number of sides, e.g. 6 for a normal die
   */
   public Die(int s)
   {
      sides = s;
      generator = new Random();
   }

   /**
      Simulates a throw of the die
      @return the face of the die
   */
   public int cast()
   {
      return 1 + generator.nextInt(sides);
   }
}
```

DieSimulator.java

```java
/**
   This program simulates casting a die ten times.
*/
public class DieSimulator
{
   public static void main(String[] args)
   {
      Die d = new Die(6);
      final int TRIES = 10;
      for (int i = 1; i <= TRIES; i++)
      {
         int n = d.cast();
         System.out.print(n + " ");
      }
      System.out.println();
   }
}
```

DieSimulator.java (cont.)

Output:

```
6 5 6 3 2 6 3 4 4 1
```

Second Run:

```
3 2 2 1 6 5 3 4 1 2
```

Self Check

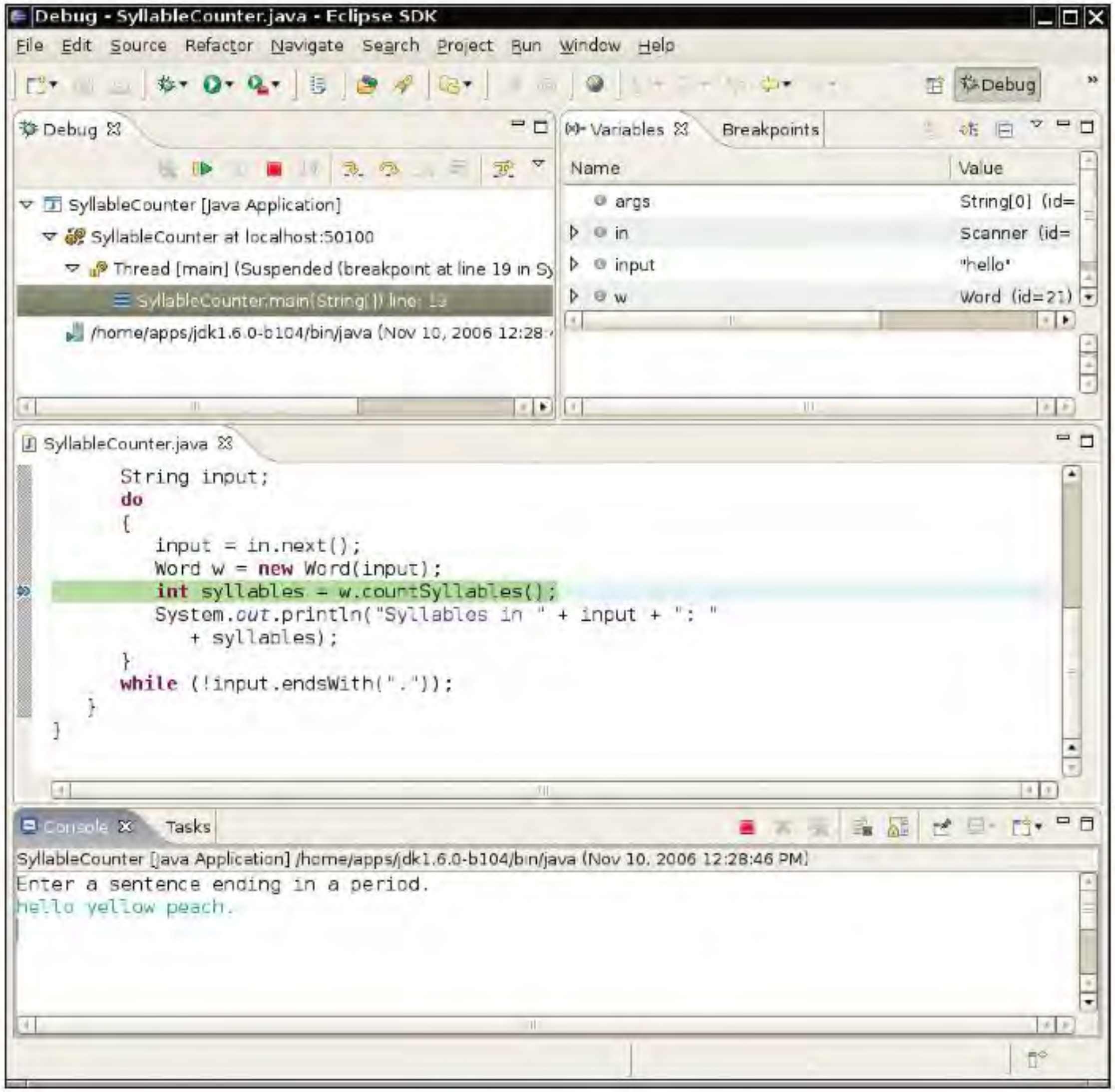
How do you use a random number generator to simulate the toss of a coin?

Answer: `int n = generator.nextInt(2); // 0 = heads, 1 = tails`

Using a Debugger

- **Debugger:** a program to execute your program and analyze its run-time behavior
- A debugger lets you stop and restart your program, see contents of variables, and step through it
- The larger your programs, the harder to debug them simply by inserting print commands
- Debuggers can be part of your IDE (e.g. Eclipse, BlueJ) or separate programs (e.g. JSwat)
- Three key concepts:
  - *Breakpoints*
  - *Single-stepping*
  - *Inspecting variables*

The Debugger Stopping at a Breakpoint


Debug - SyllableCounter.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Debug
Variables
Breakpoints
SyllableCounter.java
Console
Tasks

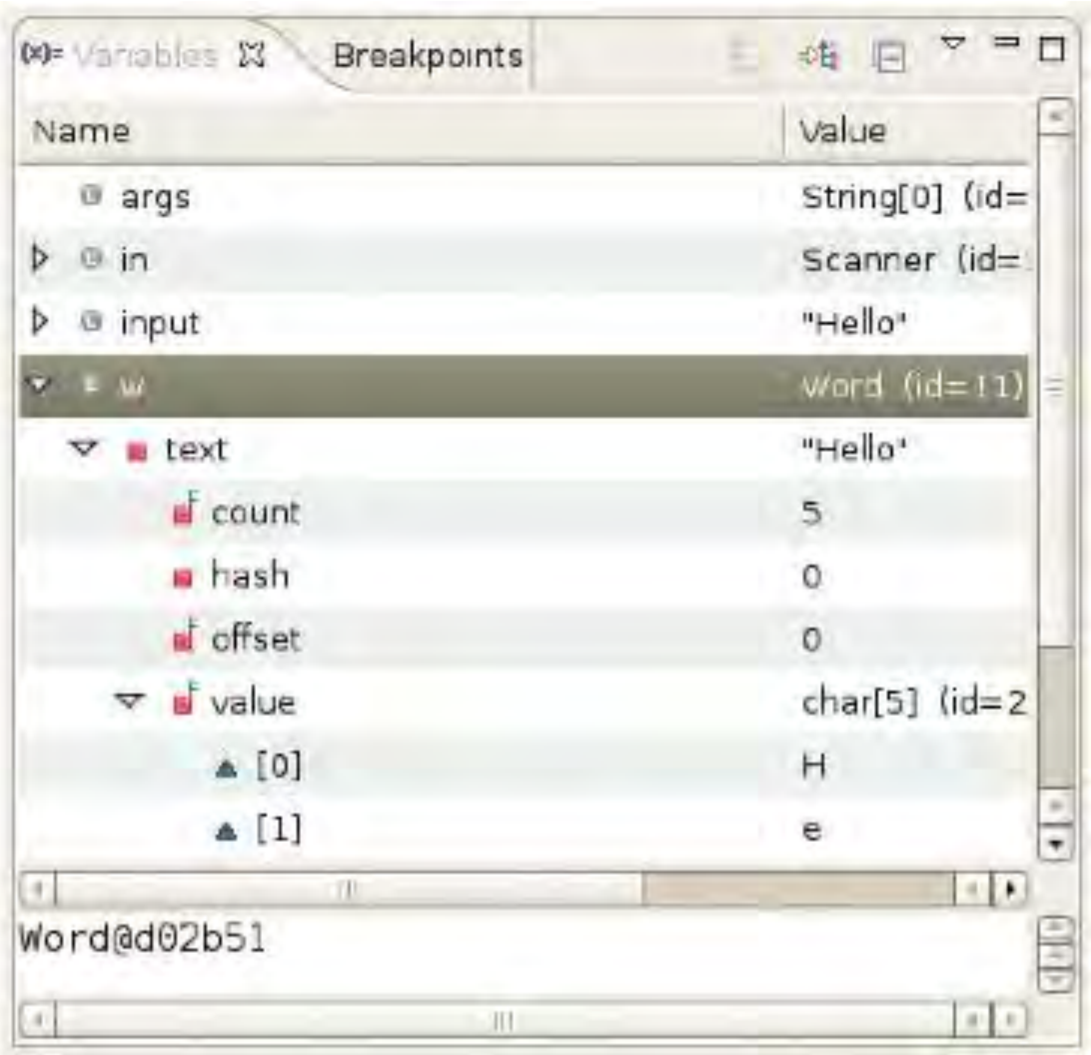
SyllableCounter [Java Application]
SyllableCounter at localhost:50100
Thread [main] (Suspended (breakpoint at line 19 in Sy
SyllableCounter.main(String[]) line: 19
/home/apps/jdk1.6.0-b104/bin/java (Nov 10, 2006 12:28

Name	Value
args	String[0] (id=
in	Scanner (id=
input	"hello"
w	Word (id=21)

```
        String input;
        do
        {
            input = in.next();
            Word w = new Word(input);
            int syllables = w.countSyllables();
            System.out.println("Syllables in " + input + ": "
                + syllables);
        }
        while (!input.endsWith("."));
    }
}
```

```
SyllableCounter [Java Application] /home/apps/jdk1.6.0-b104/bin/java (Nov 10, 2006 12:28:46 PM)
Enter a sentence ending in a period.
hello yellow peach.
```

Inspecting Variables



Name	Value
args	String[0] (id=
in	Scanner (id=
input	"Hello"
w	Word (id=11)
text	"Hello"
count	5
hash	0
offset	0
value	char[5] (id=2
[0]	H
[1]	e

```
Word@d02b51
```

Debugging

- Execution is suspended whenever a breakpoint is reached
- In a debugger, a program runs at full speed until it reaches a breakpoint
- When execution stops you can:
    - *Inspect variables*
    - *Step through the program a line at a time*
    - *Or, continue running the program at full speed until it reaches the next breakpoint*
- When program terminates, debugger stops as well
- Breakpoints stay active until you remove them
- Two variations of single-step command:
    - *Step Over: Skips method calls*
    - *Step Into: Steps inside method calls*

Single-step Example

- Current line:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " +
   syllables);
```

- When you step over method calls, you get to the next line:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " +
   syllables);
```

Single-step Example (cont.)

- However, if you step into method calls, you enter the first line of the `countSyllables` method:

```
public int countSyllables()
{
   int count = 0;
   int end = text.length() - 1;
   ...
}
```

Self Check

In the debugger, you are reaching a call to `System.out.println`. Should you step into the method or step over it?

Answer: You should step over it because you are not interested in debugging the internals of the `println` method.

Self Check

In the debugger, you are reaching the beginning of a long method with a couple of loops inside. You want to find out the return value that is computed at the end of the method. Should you set a breakpoint, or should you step through the method?

Answer: You should set a breakpoint. Stepping through loops can be tedious.

THANK YOU FOR YOUR ATTENTION !