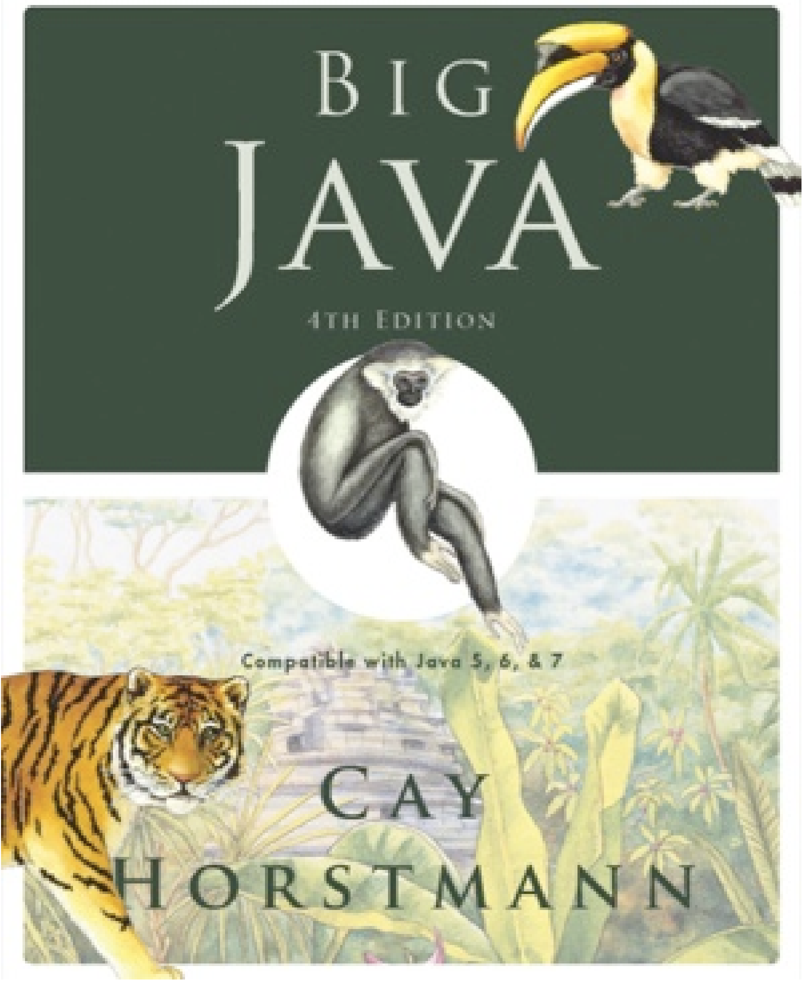


Chapter 5 – Decisions

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

T 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;
```

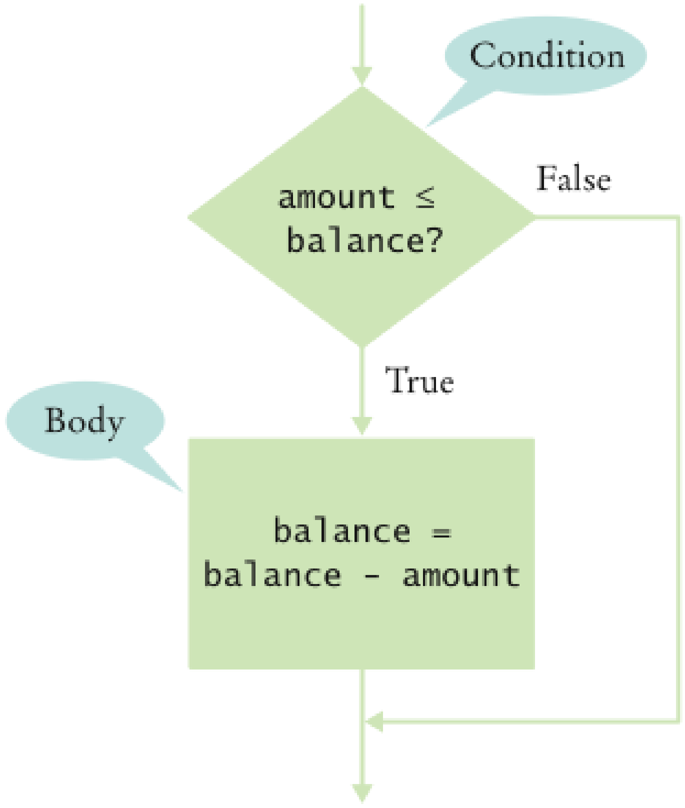


Figure 1
Flowchart for an `if` Statement

The `if/else` Statement

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

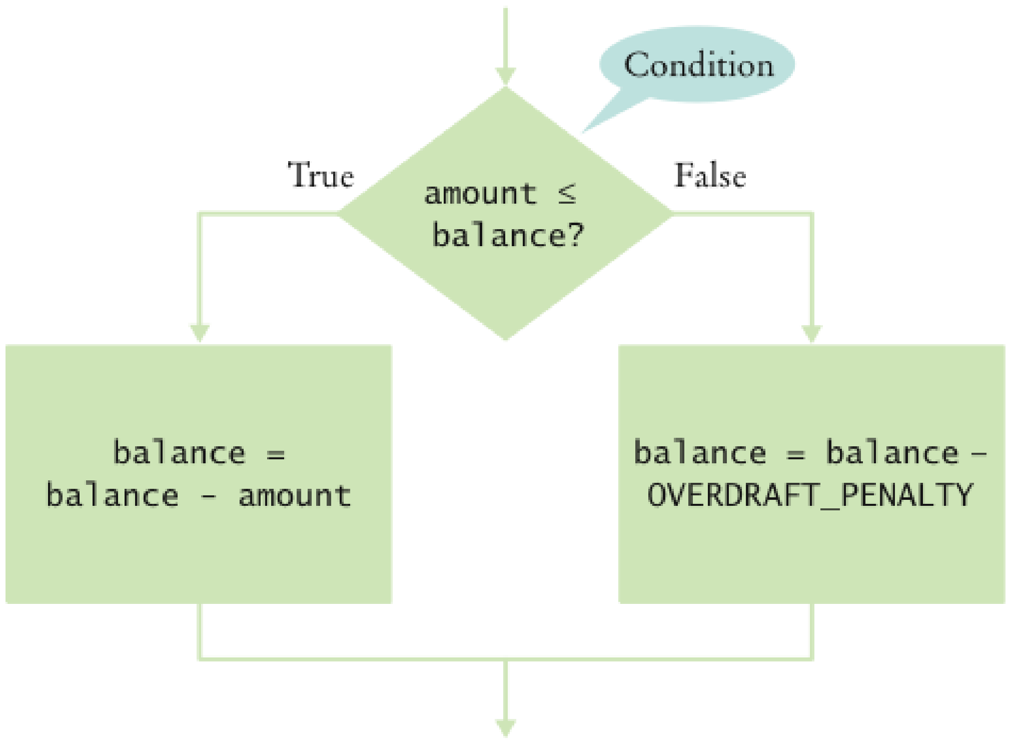

Condition
True
amount ≤ balance?
False
balance = balance - amount
balance = balance - OVERDRAFT_PENALTY

Figure 2
Flowchart for an `if/else` Statement

Statement Types

- Simple statement:

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

- Compound statement:

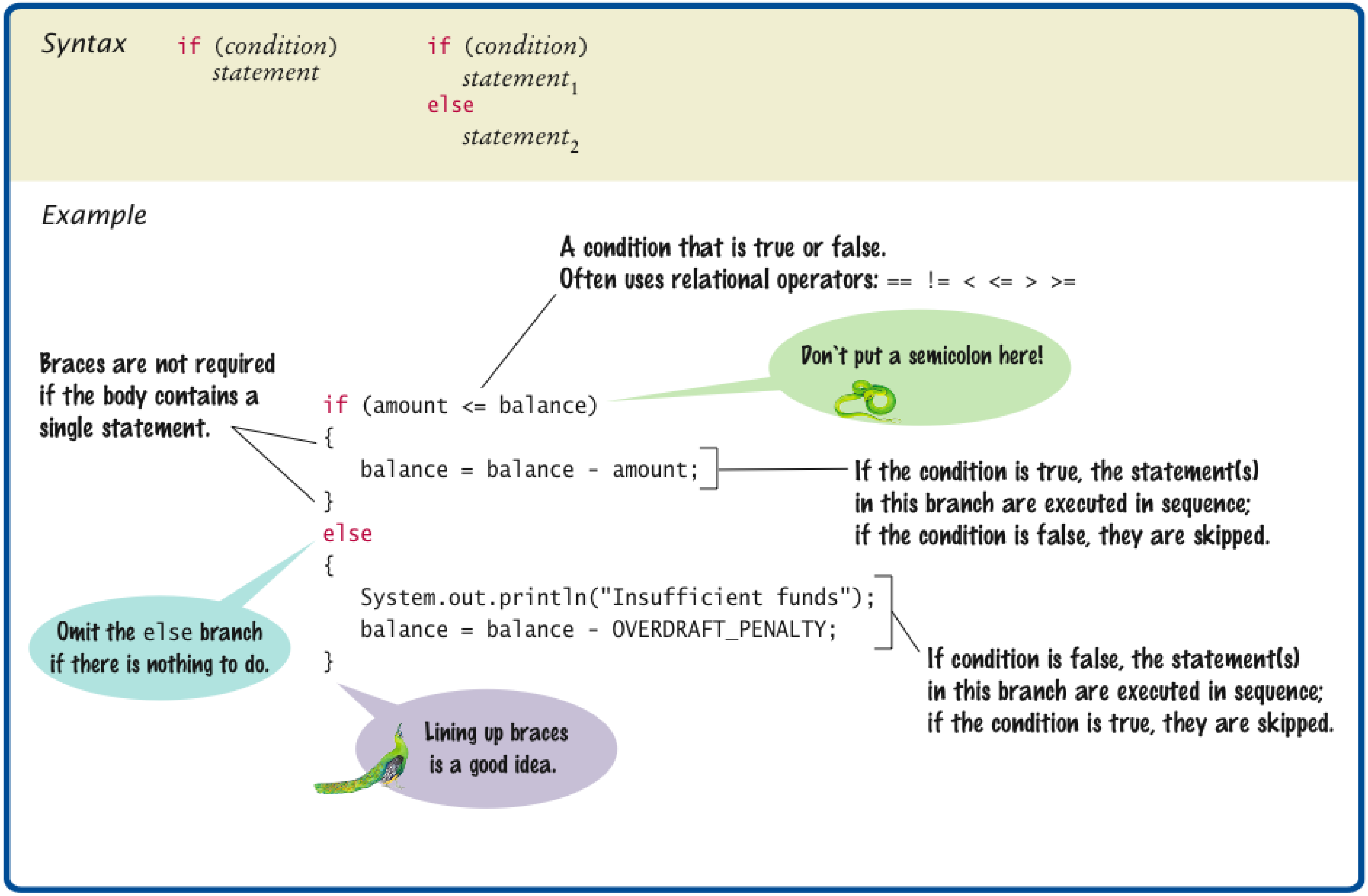
```
if (balance >= amount) balance = balance - amount;
```

Also loop statements — Chapter 6

- Block statement:

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

Syntax 5.1 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.

Comparing Values: Relational Operators

- Relational operators compare values

Java	Math Notation	Description
`>`	$>$	Greater than
`>=`	$\geq$	Greater than or equal
`<`	$<$	Less than
`<=`	$\leq$	Less than or equal
`==`	$=$	Equal
`!=`	$\neq$	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

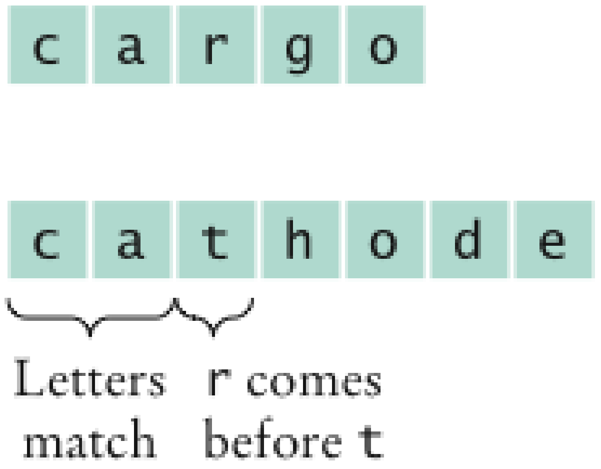
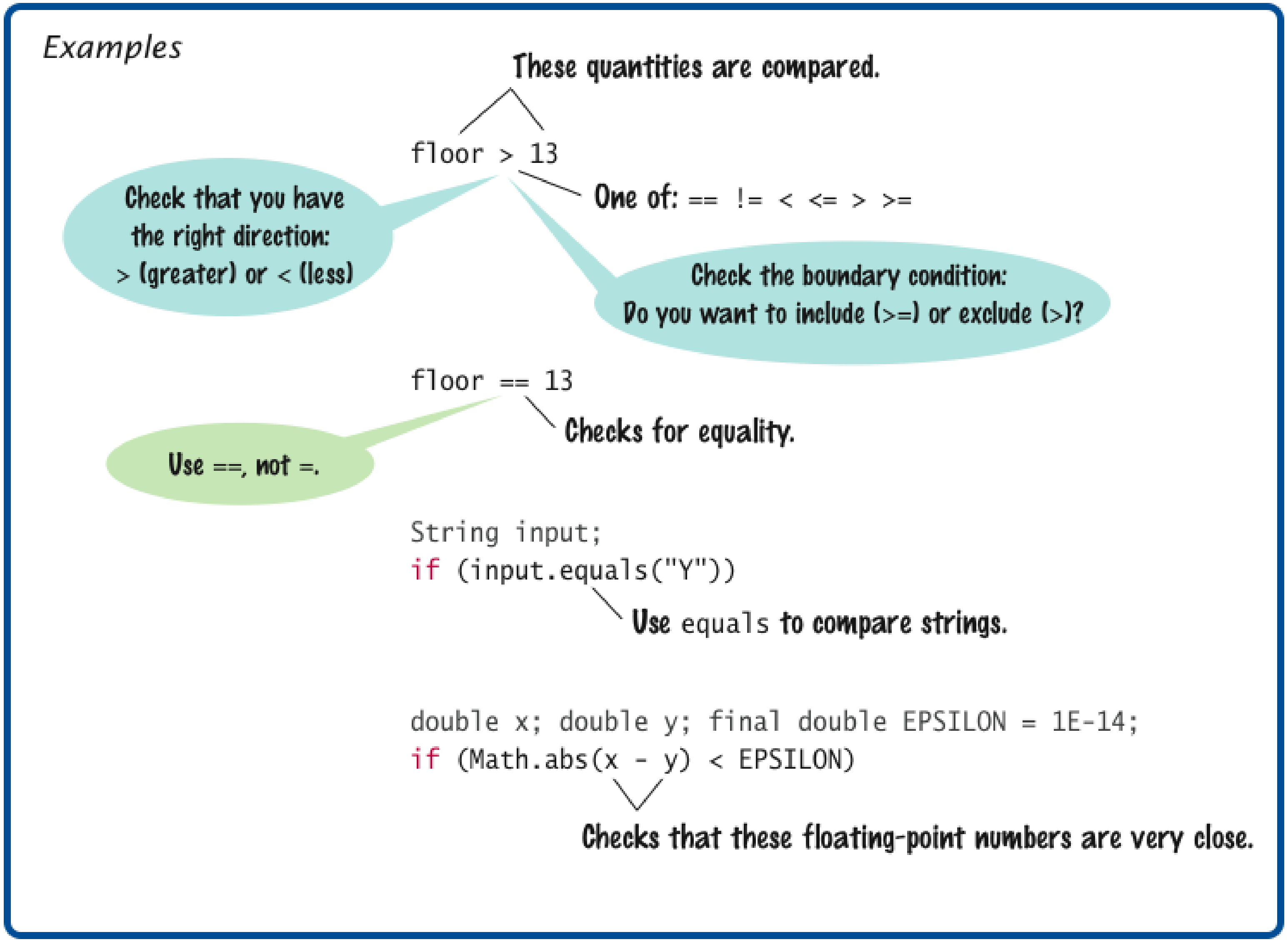


Figure 3
Lexicographic Comparison

Syntax 5.2 Comparisons



Examples

```
floor > 13

floor == 13

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

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

These quantities are compared.
One of: == != < <= > >=
Check that you have the right direction: > (greater) or < (less)
Check the boundary condition: Do you want to include (>=) or exclude (>)?
Checks for equality.
Use ==, not =.
Use equals to compare strings.
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

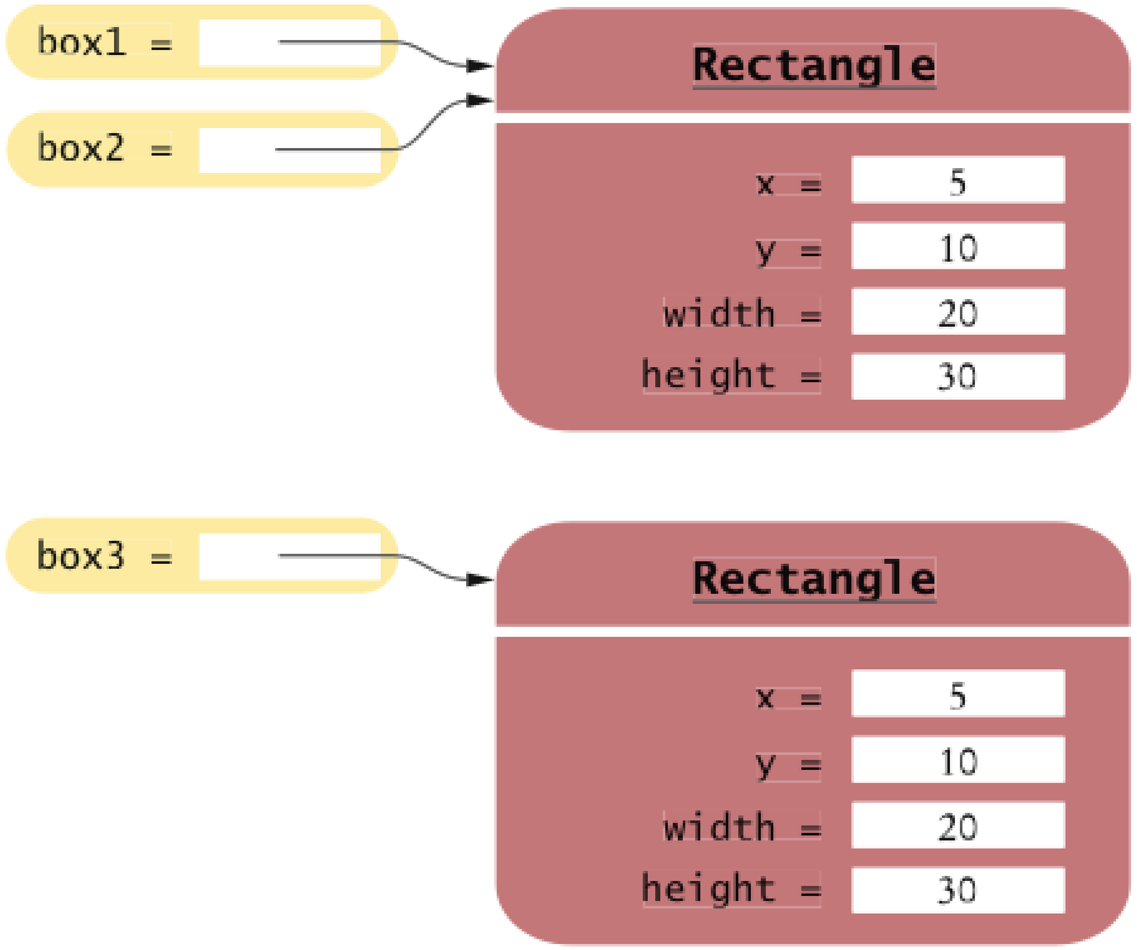


Figure 4
Comparing Object References

Let’s take a peek at toString() and equals()

`Object`: The Cosmic Superclass

- Most useful methods:
  - *`String toString()`*
  - *`boolean equals(Object otherObject)`*
  - *`Object clone()`*
- Good idea to override these methods in your classes

Overriding the `toString` Method

- Returns a string representation of the object
- Useful for debugging:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,
// height=30]"
```

- `toString` is called whenever you concatenate a string with an object:

```
"box=" + box;
// Result: "box=java.awt.Rectangle[x=5,y=10,width=20,
// height=30]"
```

Overriding the `toString` Method

- `Object.toString` prints class name and the *hash code* of the object:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to something like "BankAccount@d24606bf"
```

Overriding the `toString` Method

- To provide a nicer representation of an object, override `toString`:

```
public String toString()
{
   return "BankAccount[balance=" + balance + "]";
}
```

- This works better:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```

Overriding the `equals` Method

- `equals` tests for same *contents*:

```
if (coin1.equals(coin2)) . . .
// Contents are the same
```

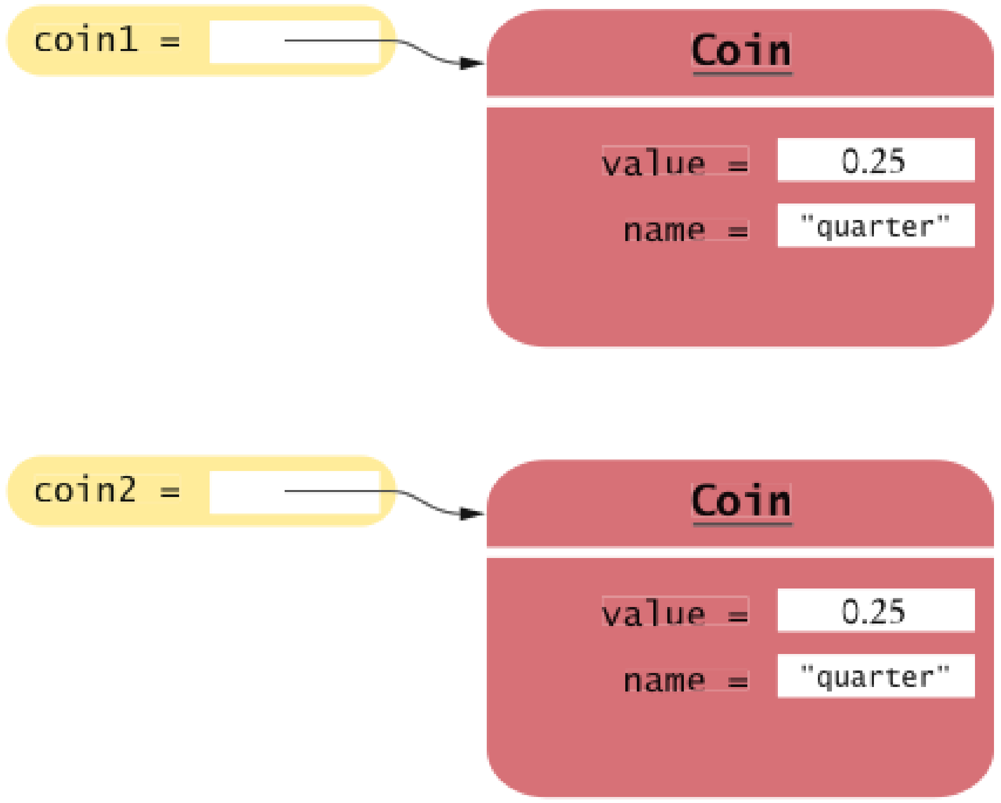


Figure 8 Two References to Equal Objects

Overriding the `equals` Method

- `==` tests for references to the same object:

```
if (coin1 == (coin2)) . . .
// Objects are the same
```

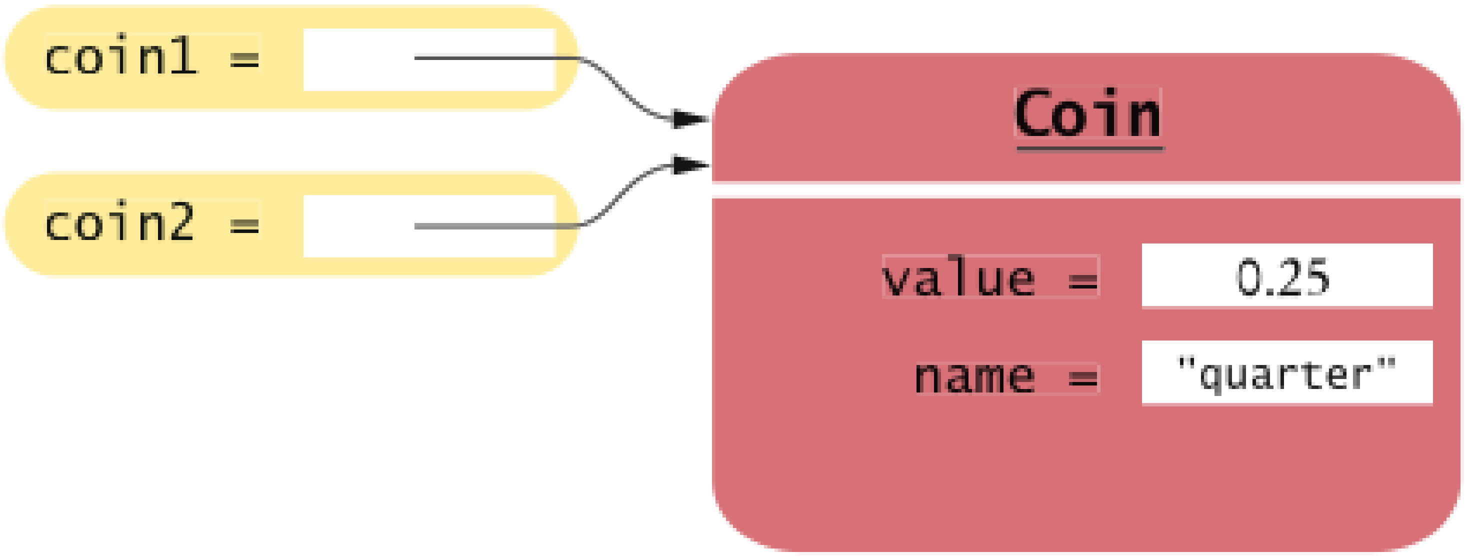


Figure 9 Two References to the Same Object

Overriding the `equals` Method

- Need to override the `equals` method of the `Object` class:

```
public class Coin
{
   ...
   public boolean equals(Object otherObject)
   {
      ...
   }
   ...
}
```

Done peeking

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

Table 1 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.

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";
```

ch05/quake/Earthquake.java

```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;
   }

```

Continued

ch05/quake/Earthquake.java (cont.)

```java
    /**
       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;
    }
 }
```

ch05/quake/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
```

The switch statement

- A sequence of if/else if/else that compares a single value against several constant alternatives can be implemented as a switch statement.

Values must be constant and can be integers, characters, enumeration constants or strings BUT not floating-point values

```
int digit;
. . .
switch (digit)
{
   case 1: System.out.print("one"); break;
   case 2: System.out.print("two"); break;
   case 3: System.out.print("three"); break;
   case 4: System.out.print("four"); break;
   case 5: System.out.print("five"); break;
   case 6: System.out.print("six"); break;
   case 7: System.out.print("seven"); break;
   case 8: System.out.print("eight"); break;
   case 9: System.out.print("nine"); break;
   default: System.out.print("error"); break;
}
```

```
int digit;
. . .
if (digit == 1) System.out.print("one");
else if (digit == 2) System.out.print("two");
else if (digit == 3) System.out.print("three");
else if (digit == 4) System.out.print("four");
else if (digit == 5) System.out.print("five");
else if (digit == 6) System.out.print("six");
else if (digit == 7) System.out.print("seven");
else if (digit == 8) System.out.print("eight");
else if (digit == 9) System.out.print("nine");
else System.out.print("error");
```

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

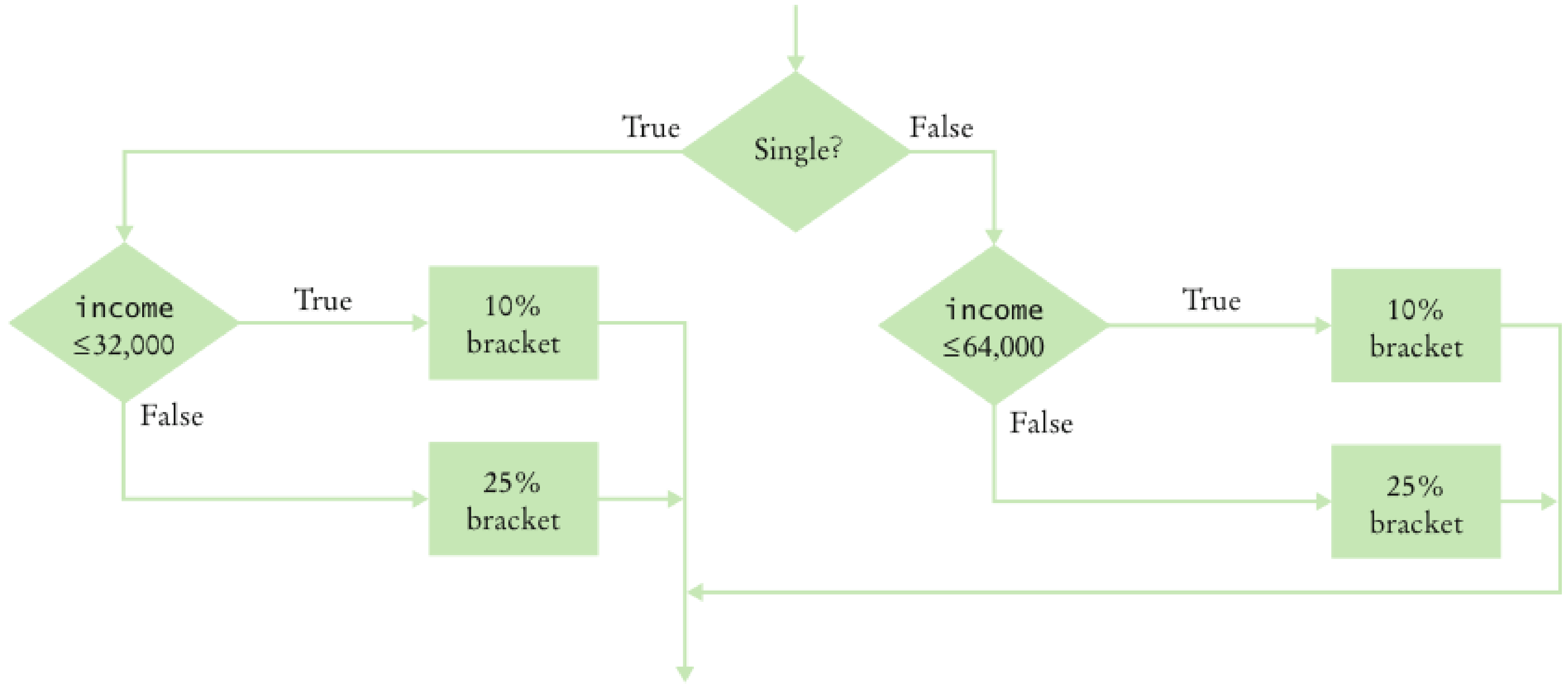


Figure 5 Income Tax Computation Using Simplified 2008 Schedule

ch05/tax/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;

```

Continued

ch05/tax/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;

```

Continued

ch05/tax/TaxReturn.java (cont.)

```java
      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;
   }
}
```

ch05/tax/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());
   }
}
```

Continued

ch05/tax/TaxCalculator.java (cont.)

Program Run:

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

Enumeration Types

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6), VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6), MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7), SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7), NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Enumeration Types

```
public static void main(String[] args) {
    double earthWeight = 175; //unit in pounds
    double mass = earthWeight/EARTH.surfaceGravity();
    for (Planet p : Planet.values())
        System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));
}
```

- The enum declaration defines a *class* (called an *enum type*). The enum class body can include data fields, constructors and methods
- The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.
- The enum values are passed to the constructor when the constant is created. Java requires that the constants be defined first, prior to any fields or methods. Also, when there are fields and methods, the list of enum constants must end with a semicolon.
- Enum has a static *values* method that returns an array containing all of the values of the enum in the order they are declared

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

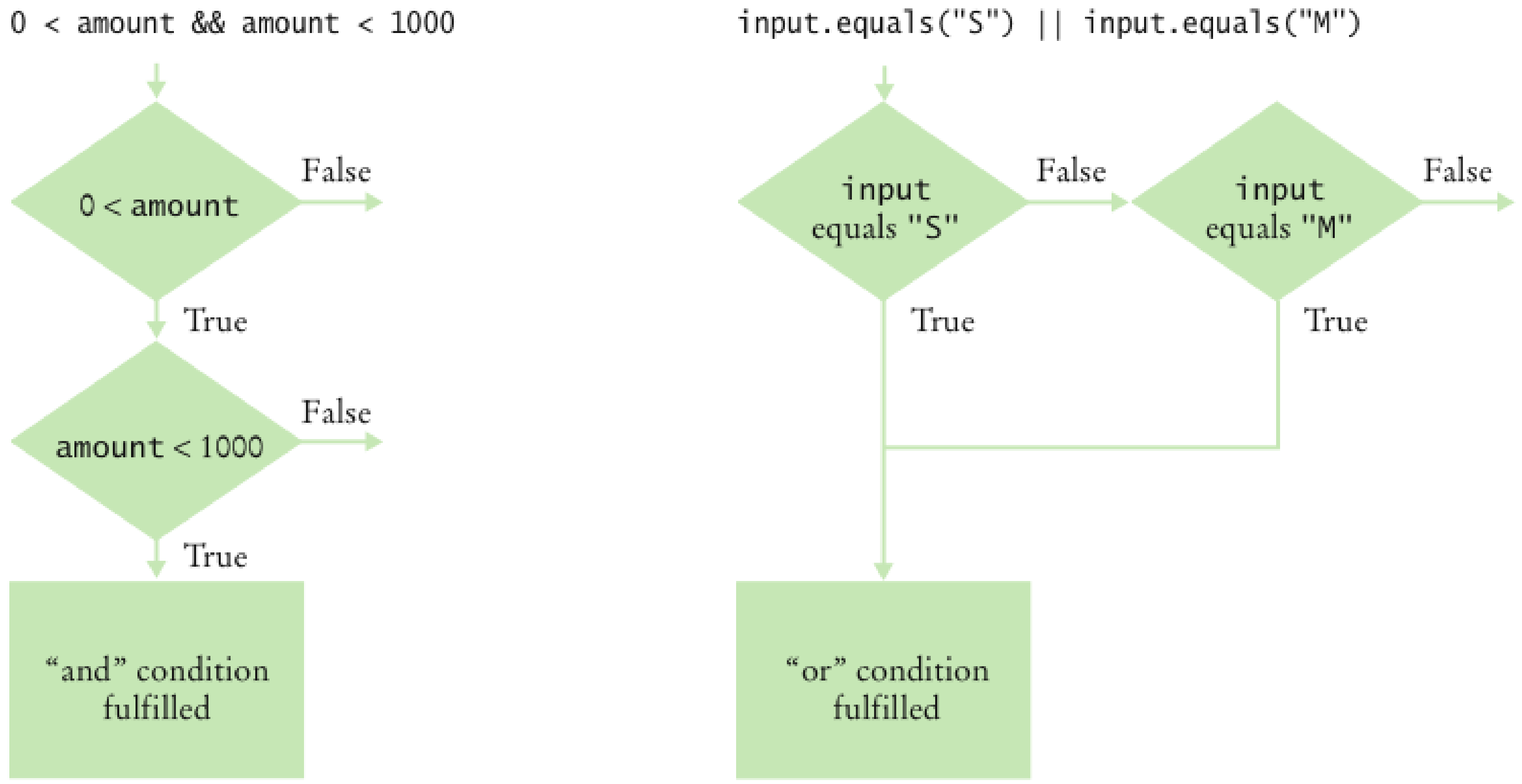


Figure 6 Flowcharts for && and || Combinations

Boolean Operators

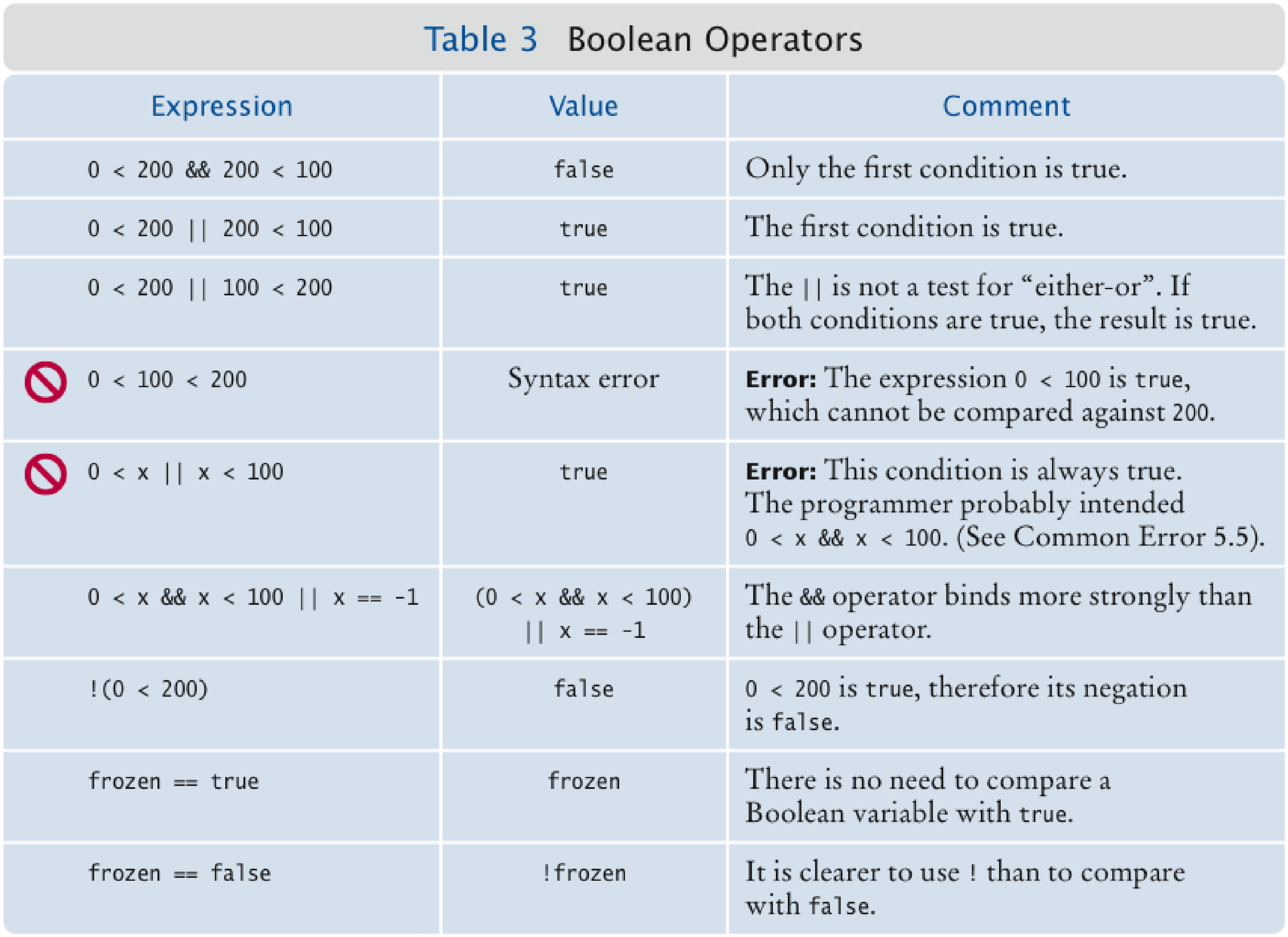


Table 3 Boolean Operators

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) ...
```

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

Code Coverage

- 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