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
The **if** Statement

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

  ```java
  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

Figure 2
Flowchart for an if/else Statement
Statement Types

• Simple statement

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

• Compound statement

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

Also

```
while, for, etc. (loop statements – Chapter 6)
```

• Block statement

```
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```
Syntax 5.1 The if Statement

if(condition)
    statement

if (condition)
    statement1

else

Example:

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

Purpose:

To execute a statement when a condition is true or false.
Syntax 5.2 Block Statement

{  
    statement_1
    statement_2
    ...
}

Example:
{  
    double newBalance = balance - amount;
    balance = newBalance;
}

Purpose:
To group several statements together to form a single statement.
Self Check 5.2

What is logically wrong with the statement

```java
if (amount <= balance) {
    newBalance = balance - amount;
    balance = newBalance;
}
```

and how do you fix it?
Comparing Values: Relational Operators

- Relational operators compare values

<table>
<thead>
<tr>
<th>Java</th>
<th>Math Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>≥</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>≤</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>==</td>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>≠</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

- The `==` denotes equality testing
  
  ```java
  a = 5; // Assign 5 to a
  if (a == 5) . . . // Test whether a equals 5
  ```
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:

\[ |x - y| \leq \varepsilon \]

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

• \( \varepsilon \) is a small number such as \( 10^{-14} \)
Comparing Strings

• Don't use == for strings!

```java
if (input == "Y") // WRONG!!!
```

• Use `equals` method:

```java
if (input.equals("Y"))
```

• `==` tests identity, `equals` tests equal contents

• Case insensitive test ("Y" or "y")

```java
if (input.equalsIgnoreCase("Y"))
```

• `s.compareTo(t) < 0` means:

  - `s` comes before `t` in the dictionary
Comparing Strings (cont.)

• "car" comes before "cargo"

• All uppercase letters come before lowercase:
  "Hello" comes before "car"
Figure 3
Lexicographic Comparison
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
Testing for null

- **null reference** refers to no object
  
  ```java
  String middleInitial = null; // Not set
  if ( . . . )
      middleInitial = middleName.substring(0, 1);
  ```

- Can be used in tests:
  ```java
  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 ""**
What is the value of `s.length()` if `s` is

- a. the empty string `""`?
- b. the string `" "` containing a space?
- c. `null`?
Multiple Alternatives: Sequences of Comparisons

if (condition1)
    statement1;
else if (condition2)
    statement2;
    ...
else
    statement4;

• 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 (cont.)

• Don't omit else

```java
if (richter >= 8.0)
    r = "Most structures fall";
if (richter >= 7.0) // omitted else--ERROR
    r = "Many buildings destroyed"
```
/**
 * A class that describes the effects of an earthquake.
 */

public class Earthquake {

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

    /**
     * Gets a description of the effect of the earthquake.
     * @return the description of the effect
     */
    public String getDescription() {
        Continued
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;

private double richter;
import java.util.Scanner;

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());
    }
}

Output:
Enter a magnitude on the Richter scale: 7.1 Many buildings destroyed
Multiple Alternatives: Nested Branches

- Branch inside another branch

```java
if (condition1)
{
    if (condition1a)
        statement1a;
    else
        statement1b;
}
else
    statement2;
```
## Tax Schedule

<table>
<thead>
<tr>
<th>If your filing status is Single</th>
<th>If your filing status is Married</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tax Bracket</strong></td>
<td><strong>Percentage</strong></td>
</tr>
<tr>
<td>$0 . . . $21,450</td>
<td>15%</td>
</tr>
<tr>
<td>Amount over $21,450, up to $51,900</td>
<td>28%</td>
</tr>
<tr>
<td>Amount over $51,900</td>
<td>31%</td>
</tr>
</tbody>
</table>
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 (cont.)

**Figure 5** Income Tax Computation Using 1992 Schedule
public class TaxReturn {

    public TaxReturn(double anIncome, int aStatus) {
        income = anIncome;
        status = aStatus;
    }

    public double getTax() {
        double tax = 0;
        if (status == SINGLE) {
            // Continued
        }
    }

}
24:      if (income <= SINGLE_BRACKET1)
25:            tax = RATE1 * income;
26:        else if (income <= SINGLE_BRACKET2)
27:            tax = RATE1 * SINGLE_BRACKET1
28:                + RATE2 * (income - SINGLE_BRACKET1);
29:        else
30:            tax = RATE1 * SINGLE_BRACKET1
31:                + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1)
32:                + RATE3 * (income - SINGLE_BRACKET2);
33:      }
34:    else
35:    {
36:        if (income <= MARRIED_BRACKET1)
37:            tax = RATE1 * income;
38:        else if (income <= MARRIED_BRACKET2)
39:            tax = RATE1 * MARRIED_BRACKET1
40:                + RATE2 * (income - MARRIED_BRACKET1);
41:        else
42:            tax = RATE1 * MARRIED_BRACKET1
43:                + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1)
44:                + RATE3 * (income - MARRIED_BRACKET2);
45:    }
47:     return tax;
48: }
49:
50:     public static final int SINGLE = 1;
51:     public static final int MARRIED = 2;
52:
53:     private static final double RATE1 = 0.15;
54:     private static final double RATE2 = 0.28;
55:     private static final double RATE3 = 0.31;
56:
57:     private static final double SINGLE_BRACKET1 = 21450;
58:     private static final double SINGLE_BRACKET2 = 51900;
59:
60:     private static final double MARRIED_BRACKET1 = 35800;
61:     private static final double MARRIED_BRACKET2 = 86500;
62:
63:     private double income;
64:     private int status;
65: }
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;
    }
}
ch05/tax/TaxCalculator.java (cont.)

23:       TaxReturn aTaxReturn = new TaxReturn(income, status);
24:       
25:       System.out.println("Tax: ");
26:          + aTaxReturn.getTax());
27:       
28:       }

Output:
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
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
• 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")) . . .`
&& and || Operators

Figure 6  Flowcharts for && and || Combinations
## Truth Tables

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>Any</td>
<td>false</td>
</tr>
</tbody>
</table>

| A    | B    | A || B |
|------|------|-------|
| true | Any  | true  |
| false| true | true  |
| false| false| false |

<table>
<thead>
<tr>
<th>A</th>
<th>! A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
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 5.7

When does the statement

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

print false?