Chapter Goals

- To implement decisions using the if statement
- To compare integers, floating-point numbers, and Strings
- To write statements using the Boolean data type
- To develop strategies for testing your programs
- To validate user inputs

In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.

Contents

- The if Statement
- Comparing Numbers and Strings
- Multiple Alternatives
- Nested Branches
- Problem Solving: Flowcharts
- Problem Solving: Test Cases
- Boolean Variables and Operators
- Application: Input Validation

3.1 The if Statement

- A computer program often needs to make decisions based on the input, or circumstances
- For example, buildings often ‘skip’ the 13th floor, and elevators should too
  - The 14th floor is really the 13th floor
  - So every floor above 12 is really ‘floor - 1’
    - If floor > 12, Actual floor = floor - 1
- The two keywords of the if statement are:
  - if
  - else

The if statement allows a program to carry out different actions depending on the nature of the data to be processed.
An if statement lets a program carry out actions only when the condition is met.

\[
\text{if (amount} \leq \text{balance)}
\]

- balance = balance - amount;

---

Tip On Using Braces

- Use braces for block statements:

\[
\text{if (amount} \leq \text{balance)}
\]

- It is a good idea to use braces even there is only one statement in the if-branch/else-branch to make the statement more readable.

---

Common Error 3.1

A semicolon after an if statement:

\[
\text{if (floor} > 13) ;
\]

- The if statement has an empty body.
- The ‘body’ (between the curly braces) will always be executed in this case

\[
\begin{align*}
\text{floor} &= 1; \text{ what is that value of floor after the if-statement?} \\
\text{floor} &= 15; \text{ what is the value of floor after the if-statement?}
\end{align*}
\]
The Conditional Operator

- Short-hand: Condition True branch False branch
  - Includes all parts of an if-else clause, but uses:
    - `?` To begin the true branch
    - `:` To end the true branch and start the false branch
  
  ```java
  actualFloor = floor > 13 ? floor - 1 : floor;
  ```

3.2 Comparing Numbers and Strings

- Every `if` statement has a condition.
  - Relational operator used to compares two integers:

  ```
<table>
<thead>
<tr>
<th>Table 1</th>
<th>Relational Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Math Notation</td>
</tr>
<tr>
<td>if (floor &gt; 13) ..</td>
<td>&gt;</td>
</tr>
<tr>
<td>if (floor &gt;= 13) ..</td>
<td>≥</td>
</tr>
<tr>
<td>if (floor &lt; 13) ..</td>
<td>&lt;</td>
</tr>
<tr>
<td>if (floor &lt;= 13) ..</td>
<td>≤</td>
</tr>
<tr>
<td>if (floor == 13) ..</td>
<td>=</td>
</tr>
<tr>
<td>!=</td>
<td>≠</td>
</tr>
</tbody>
</table>
  ```

Operator Precedence

- The comparison operators have lower precedence than arithmetic operators.
  - Arithmetic operations are done before the comparison

  ```java
  if (floor > height + 1)
  ```

Relational Operators

```java
<table>
<thead>
<tr>
<th>Table 2</th>
<th>Relational Operator Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Value</td>
</tr>
<tr>
<td>3 &lt; 4</td>
<td>true</td>
</tr>
<tr>
<td>3 &lt;= 4</td>
<td>Error</td>
</tr>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>4 &lt; 4</td>
<td>false</td>
</tr>
<tr>
<td>4 &lt;= 4</td>
<td>true</td>
</tr>
<tr>
<td>3 == 5 - 2</td>
<td>true</td>
</tr>
<tr>
<td>3 != 5 - 1</td>
<td>true</td>
</tr>
</tbody>
</table>
```
## Common Error 3.2

### Comparison of Floating-Point Numbers
- Floating-point numbers have **limited precision**
- **Round-off errors** can lead to unexpected results

```java
double r = Math.sqrt(2.0);
if (r * r == 2.0)
{
    System.out.println("Math.sqrt(2.0) squared is 2.0");
}
else
{
    System.out.println("Math.sqrt(2.0) squared is not 2.0 but " + r * r);
}
```

**Output:**
```
Math.sqrt(2.0) squared is not 2.0 but 2.00000000000000044
```

## Compare floating point numbers

- Use a very small value to compare the difference if floating-point values are ‘*close enough*’
  - The magnitude of their difference should be less than some threshold
  - Mathematically, we would write that \( x \) and \( y \) are close enough if:
    \[
    \left| x - y \right| < \varepsilon
    \]

```java
final double EPSILON = 1E-14;
double r = Math.sqrt(2.0);
if (Math.abs(r * r - 2.0) < EPSILON)
{
    System.out.println("Math.sqrt(2.0) squared is approx. 2.0");
}
```

## Comparing Strings

- Do not use the `==` to compare Strings
  - `==` compares the **locations** of two strings, and not their **contents**

```java
if ("Harry" == "Harry") //true
    String name1="Harry";
    String name2="Harry";
    if(name1==name2) // false, because name1 and name2 are two different String objects, even they contain the same value "Harry"
```
Comparing Strings using `equals`

- `equals` tests equal contents
  ```java
  String name1="Harry";
  String name2="Harry";
  if(name1.equals(name2)) //true
  ```

- For case insensitive test ("Y" or "y"), use `equalsIgnoreCase`
  ```java
  if (input.equalsIgnoreCase("Y"))
  ```

---

Lexicographical Order

- Space comes before numbers:
  - " 45" comes before "1"
  - " 45".compareTo("1")?

- Numbers come before upper case letters:
  - "1" comes before "Hello"
  - "1".compareTo("Hello")?

- Upper case letters come before lower case letters:
  - "Hello" comes before "car"
  - "Hello".compareTo("car")?

---

Comparison Summary

**Examples**

```
// These quantities are compared
int a = 10; int b = 20; int c = 15;

// Integer types
if (a < b)
  System.out.println("a < b");
else
  System.out.println("a >= b");

// Floating-point types
double x = 3.14; double y = 3.14159;

// String types
String s1 = "Hello World";
String s2 = "World Hello";

if (s1.equalsIgnoreCase(s2))
  System.out.println("s1 == s2");
else
  System.out.println("s1 != s2");
```
The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than $128, and a 16 percent discount if the price is at least $128.

```
if (originalPrice < 128) {
    discountRate = 0.92;
} else {
    discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```

What if you have more than two possibilities?

The earthquake effect example:

- scale $\geq 8.0$
- $7.0 \leq scale < 8.0$
- $6.0 \leq scale < 7.0$
- $4.5 \leq scale < 6.0$
- scale $< 4.5$

```
if (richter $\geq 8.0$) // Handle the 'special case' first
    System.out.println("Most structures fall");
} else if (richter $\geq 7.0$)
    System.out.println("Many buildings destroyed");
} else if (richter $\geq 6.0$)
    System.out.println("Many buildings considerably damaged, some collapse");
} else if (richter $\geq 4.5$)
    System.out.println("Damage to poorly constructed buildings");
} else // so that the 'general case' can be handled last
    System.out.println("No destruction of buildings");
```
What is wrong with this code?

```java
if (richter >= 8.0) {
    System.out.println("Most structures fall");
} else if (richter >= 7.0) {
    System.out.println("Many buildings destroyed");
} else if (richter >= 6.0) {
    System.out.println("Many buildings damaged, some collapse");
} else if (richter >= 4.5) {
    System.out.println("Damage to poorly constructed buildings");
}
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Alternative Implementation

```java
if (richter < 4.5) // Handle the 'special case' first
    System.out.println("No destruction of buildings");
else if (richter < 6.0)
    System.out.println("Damage to poorly constructed buildings");
else if (richter < 7.0)
    System.out.println("Many buildings damaged, some collapse");
else if (richter < 8.5)
    System.out.println("Many buildings destroyed");
else // so that the 'general case' can be handled last
    System.out.println("Most structures fall");
```

Copyright © 2011 by John Wiley & Sons. All rights reserved.

Another way to multiway branch

- The `switch` statement chooses a `case` based on an [primitive type](#) value (In Java 7, String value also works)
- **Primitive types:** `int`, `char`, `short`, `byte`
- `break` ends each `case`
- `default` catches all other values

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Switch without break

- **If the `break` is missing, the case falls through to the next case's statements.**
  - `digit = 1`, `digitName`?
  - `digit = 6`, `digitName`?
  - `digit=-111`, `digitName`?

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

Copyright © 2011 by John Wiley & Sons. All rights reserved.
The Switch Statement

The type of the Expression in the Switch statement can be:
- char
- byte
- short
- int
- enum (later)

```java
char digit = ...;
switch (digit)
{
    case '1':
        digitName = "one";
        break;
    case '2':
        digitName = "two";
        break;
    case '3':
        digitName = "three";
        break;
    case '4':
        digitName = "four";
        break;
    case '5':
        digitName = "five";
        break;
    case '6':
        digitName = "six";
        break;
    case '7':
        digitName = "seven";
        break;
    case '8':
        digitName = "eight";
        break;
    case '9':
        digitName = "nine";
        break;
    default:
        digitName = "";
        break;
}
```

More on switch

```java
char c = ...;
switch (c)
{
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    case '0':
        System.out.println("c is digit");
        break;
    case '\n':
    case '\t':
        System.out.println("c is space");
        break;
    default:
        System.out.println("c is other");
        break;
}
```

3.4 Nested if-else statement

You can nest an if inside either branch of an if statement.

Example: Ordering drinks
- Ask the customer for their drink order
- if customer orders alcoholic drink
  - Ask customer for ID
  - if customer’s age is 21 or over
    - Serve alcoholic drink
  - else
    - Politely explain the law to the customer
- else
  - Serve customers a non-alcoholic drink

Flowchart of a Nested if

- Nested if-else inside true branch of an if statement.
  - Three paths
Calculate Income Tax Return

<table>
<thead>
<tr>
<th>If your filing status is Single</th>
<th>If your filing status is Married</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tax Bracket</td>
<td>Tax Bracket</td>
</tr>
<tr>
<td>Percentage</td>
<td>Percentage</td>
</tr>
<tr>
<td>$0 ... $32,000</td>
<td>0 ... $64,000</td>
</tr>
<tr>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>Amount over $32,000</td>
<td>Amount over $64,000</td>
</tr>
<tr>
<td>25%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Testing multiple variables
- filing status: single or married
- Income:

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

Flowchart for Tax Example
- Four branches

Input:
- Testing filing status
- Testing income

Output:
- If filing status is Single:
  - Income ≤ $32,000: 10% bracket
  - Income > $32,000: 25% bracket
- If filing status is Married:
  - Income ≤ $64,000: 10% bracket
  - Income > $64,000: 25% bracket

TaxCalculator.java
```java
import java.util.Scanner;

class TaxCalculator {
    public static void main(String[] args) {
        double tax1 = 0;
        double tax2 = 0;
        // Read income and marital status
        Scanner in = new Scanner(System.in);
        double income = in.nextDouble();
        System.out.print("Please enter your income: ");
        System.out.print("Please enter s for single, m for married: ");
        String maritalStatus = in.nextLine();
```

Page 40
The ‘True’ branch (Single)
   - Two branches within this branch

```java
// Compute taxes due
if (maritalStatus.equals("s")) {
    if (income <= RATE1_SINGLE_LIMIT) {
        tax1 = RATE1 * income;
    } else {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
}
```

The ‘False’ branch (Married)

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

Program Run

```
Please enter your income: 8000
Please enter s for single, m for married: m
The tax is 10400
```

Common Error 3.4

The Dangling else Problem

- When an if statement is nested inside another if statement, the else clause is always associated with the closest if

```java
int a = ...; int b = ...;
if (a > 5) {
    if (b > 5 )
        System.out.println("a and b are > 5");
    else
        System.out.println("a is > 5");
}
```

To avoid confusion, use bracket for blocked if statements

3.6 Test Cases for if-else

- Aim to complete coverage of all branches:
  - There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases

Each branch of your code should be covered with a test case
Choosing Test Cases

- Choose **input values** in the condition statement:
  - For each branch, test **boundary conditions**, such as an income that is at the boundary between two tax brackets, and a zero income
  - If you are responsible for error checking (which is discussed in Section 3.8), also test an invalid input, such as a negative income

<table>
<thead>
<tr>
<th>Test Case</th>
<th>status</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>single</td>
<td>32,000</td>
</tr>
<tr>
<td>2</td>
<td>single</td>
<td>33,000</td>
</tr>
<tr>
<td>3</td>
<td>married</td>
<td>64,000</td>
</tr>
<tr>
<td>4</td>
<td>married</td>
<td>65,000</td>
</tr>
</tbody>
</table>

3.7 The Boolean Type

- **Boolean Values**
  - **true** or **false**

- **Boolean Variables**
  - `boolean` is a Java data type
    - `boolean` found;

- **Boolean Operators**: `&& and` `|| or` `! not`
  - `&&` is the **and** operator
  - `||` is the **or** operator
  - `!` is the **not** operator

- **and operator**: `&&`

  - Both sides of the `&&` must be true for the result to be true:
    ```java
    if ((interest = amount * rate) > 100 &&
        (payment = amount / 12) > 200) {...} // use interest and payment to compute something
    ```
  - Lazy evaluation: if A is false, return false without evaluating B
  - When there is need to evaluate both A and B, use & instead of `&&`

- **or operator**: `||`

  - Only one of the two conditions has to be true for the statement to be true
    ```java
    if ((interest = amount * rate) > 100 ||
        (payment = amount / 12) > 200) {...} // use interest and payment to compute something
    ```
  - Lazy evaluation: if A is true, return true without evaluating B
  - When there is need to evaluate both A and B, use | instead of `||`
Convert between switch & if

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

Convert between if & switch

```java
int a = . . .;
if (a > 0 && a <= 3)
    System.out.println("small");
else if (a > 3 && a <=6)
    System.out.println("medium");
else if (a > 6 && a < 10)
    System.out.println("large");
else
    System.out.println("wrong size");
```

The not Operator: !

- The NOT operator inverts the value of a boolean variable:

```
if (!attending || grade < 60)
    System.out.println("Drop?");
```

- It is considered gauche to write a test such as:

```
if (attending == false || (grade < 60) == true)
```

Predicate Methods

- A method that returns a boolean value is called a predicate method.
- The Character class has a number of predicate methods:

```
table 5 Character Testing Methods
<table>
<thead>
<tr>
<th>Method</th>
<th>Examples of Accepted Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDigit</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>isLetter</td>
<td>A, B, C, a, b, c</td>
</tr>
<tr>
<td>isUpperCase</td>
<td>A, B, C</td>
</tr>
<tr>
<td>isLowerCase</td>
<td>a, b, c</td>
</tr>
<tr>
<td>isWhitespace</td>
<td>space, newline, tab</td>
</tr>
</tbody>
</table>
```
### Boolean Operator Examples

#### Table 5: Boolean Operator Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt;= 200 &amp;&amp; 200 &lt;= 100</td>
<td>false</td>
<td>Only the first condition is true.</td>
</tr>
<tr>
<td>0 &lt; 200</td>
<td></td>
<td>100 &lt; 200</td>
</tr>
<tr>
<td>0 &lt; 200</td>
<td></td>
<td>100 &lt; 200</td>
</tr>
<tr>
<td>0 &lt; x &amp;&amp; x &lt; 200</td>
<td></td>
<td>x &gt;= -1 (0 &lt; x &amp;&amp; x &lt; 200)</td>
</tr>
<tr>
<td>0 &lt; x &lt; 100</td>
<td>Error</td>
<td>Error: This expression does not test whether x is between 0 and 100. The expression 0 &lt; x is a Boolean value. You cannot compare a Boolean value with the integer 100.</td>
</tr>
</tbody>
</table>

### Common Error 3.6

**Confusing && and || Conditions**

- A value \( x \) lies between \( a \) and \( b \) if it is at least a **and** at most \( b \).
- A value \( y \) lies outside that range if it is less than a **or** greater than \( b \).

### Lazy Evaluation: &&

- Combined conditions are evaluated from left to right
  - If the left half of an **and** condition is false, why look further?

```java
if (temp > 0 && temp < 100)
{
    System.out.println("Liquid");
}
```

- A useful example:

```java
if (quantity > 0 && price / quantity < 10)
```

---

*Page 69*
if (temp <= 0 || temp >= 100) {
    System.out.println("Not Liquid");
}

### Lazy Evaluation

- If the left half of the or is true, why look further?

- Java doesn’t!
- Don’t do these second:
  - Assignment
  - Output

### 3.8 Input Validation

- **Input Validation:**
  - A user input may have an invalid character or value (wrong type).
  - Use Scanner methods to validate user inputs.
    - `hasNextInt`:
      - True if integer
      - False if not
  - Then range check value:
  - We expect a floor number to be between 1 and 20
    - NOT 0, 13 or > 20

```java
if (in.hasNextInt()) {
    int floor = in.nextInt();
    // Process the input value
    if (floor == 13) {
        System.out.println("Error: There is no thirteenth floor.");
    } else if (floor <= 0 || floor > 20) {
        System.out.println("Error: The floor must be between 1 and 20.");
    } else {
        System.out.println("The elevator will travel to the actual floor =
                          + actualFloor");
    }
} else {
    System.out.println("Not integer.");
}
```
Summary: if Statement

- The if statement allows a program to carry out different actions depending on the nature of the data to be processed.
- Relational operators (< <= > >= == !=) are used to compare numbers and Strings.
- Do not use the == operator to compare Strings.
  - Use the equals method instead.
  - The compareTo method compares Strings in lexicographic order.
- Multiple if statements can be combined to evaluate complex decisions.
- When using multiple if statements, test general conditions after more specific conditions.

Summary: Nested IF

- When a decision statement is contained inside the branch of another decision statement, the statements are nested.
- Nested decisions are required for problems that have two levels of decision making.
- Flow charts are made up of elements for tasks, input/output, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.
- Each branch of your program should be covered by a test case.
- It is a good idea to design test cases before implementing a program.

Summary: Boolean

- The Boolean type boolean has two values, true and false.
  - Java has two Boolean operators that combine conditions: && (and) and || (or).
  - To invert a condition, use the ! (not) operator.
  - The && and || operators are computed lazily: As soon as the false (for &&) or truth (for ||) value is determined, no further condition is evaluated.
- You can use Scanner hasNext methods to ensure that the data is what you expect.