Chapter 6 - Decisions

Characters
- **char**
  - primitive data type
  - stores a single character
  - operations (same as int):
    - + -
  - special chars (use backslash):
    - ‘\n’ ‘\t’ ‘\’ ‘\’ ‘\’

Example
  - char c = ‘a’;
  - char q = ‘\’; // stores a single quote

Practice
- int i = (int) 'D' 68
- char c = (char) 106 79
- i = 'D' + 'A' 133
- c = (char)('D' + 42) 'n'

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

Control
- The real world requires decisions
  - Is it time to feed the cat? Can a book be checked out?
  - Essential feature of nontrivial programs is to make decisions based on input
- The answers to these decision require different actions
- Introduces the need for control statements

Boolean
- **boolean** is a primitive data type that holds one of two values
  - true
  - false

- Any time a question needs to be asked, we use a boolean expression, which returns a boolean value
  - As as opposed to a mathematical expression which returns a number value
Relational Operators

- What operators are used for boolean expressions?

  `<`  less than
  `<=`  less than or equal to
  `>`  greater than
  `>=`  greater than or equal to
  `==`  equal to
  `!=`  not equal to

Boolean Operators

- Boolean expressions are just like mathematical expressions
- But here, they return a true or false (not int, double, etc)

- These are the 6 binary operators

  - Ex.
    - `testScore < 80`
    - `testScore * 2 > 350`

if statement

- Conditional statements are represented with if-else statements

  ```java
  Scanner stdin = new Scanner(System.in);
  int testScore = stdin.nextInt();
  if (testScore < 70){
    System.out.println("You are below the mean");
  } else {
    System.out.println("You are above the mean");
  }
  ```

if statements

- When the code reaches the if statement, it evaluates the boolean expression

- If it is true, the <then block> is executed

- If it is false, the <else block> is executed

- Called a branching statement because it branches to a block of code

BankAccount Revisited

- Withdraw method we implemented allowed user to withdraw as much money as they wanted
- Is this realistic?

- What decision should `withdraw()` make?

- What should the decision be made on?
Version 2.0

```java
public void withdraw(double amount) {
    if (amount <= balance) {
        balance -= amount;
    } else {
        balance = balance - OVERDRAFT_PENALTY;
    }
}
```

Figure 1
Flowchart for an if Statement

Simple vs. Compound

- The decision we have made only resulted in one instruction
- Most decisions lead to a path of instructions, requiring multiple statements. The statements are grouped with curly braces {} if (amount <= balance) {
  System.out.println("Legal Withdrawal");
  balance = balance - amount;
} else {
  balance = balance - OVERDRAFT_PENALTY;
}

Figure 2
Flowchart for an if/else Statement

Productivity Hint 6.1
- Style guide – use indentation to indicated nesting levels
  public class BankAccount
  {
    public void withdraw()
    {
      if (amount <= balance) {
        balance -= amount;
      }
    }
  }

If statement
- If you have a single statement, {} not required
  if (testScore < 70)
      System.out.println("You are below the mean");
  else
      System.out.println("You are above the mean");
- But convention to use them anyways
  - Helpful for adding temporary output to check if a branch is executed
  - Makes nested if-else statements easier to read (later)
  - Avoid errors
**Definition**

- **Simple Statement**
  - Basic statement such as `balance = balance - amount;`
- **Block statement** — a group of statements enclosed within curly braces
  - Methods
  - Classes
  - If statements, Switch, Loops
- **Compound statement**
  - Statements that have multiple paths of execution
    - If statements, loops

**Syntax – if statement**

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

*statement* must be a statement - simple, compound, or block

---

**Comparing integers**

- Comparing whole numbers is trivial

```java
int a = 5;
if (a == 5){
    System.out.println("Wow this is easy!");
} else {
    System.out.println("I wanna go home!");
}
```

---

**Comparing Floating Point**

- Comparing real numbers is more difficult when taking into account roundoff errors.

```java
double r = Math.sqrt(2);
double d = r * r - 2;
if (d != 0){
    System.out.println("Your equation is incorrect, the result is " + d);
}
```

Your equation is incorrect, the result is 4.440892098500626E-16

---

**Roundoff**

- This is a problem, since logically the equation makes sense
- Lesson: Floating point numbers may not be exact
- Solution: Use range of values that can be valued
  - Should be close to 0

- 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| ≤ ε`

```java
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}
if Statement

- In an if statement, the block {} is only executed if the condition expression is true.

```
if (x == y)
{
    System.out.println("x equals y");
    // Indent stmts inside the block.
}
```

Example Setup

```
// Declare and create Die objects
Die die1 = new Die();
Die die2 = new Die();

// Roll die objects
die1.roll();
die2.roll();

// Save results in new variables
int roll1 = die1.getTop();
int roll2 = die2.getTop();
```

Simple if Statement

```
// Test for doubles
if (roll1 == roll2)
{
    System.out.println("Doubles");
}
// rest of program
```

Control Flow of If statement

```
if (roll1 == roll2)
{
    System.out.println("Doubles");
    // continue with rest of program
}
else
{
    System.out.println("Sorry, no doubles");
    // continue with rest of program
```

Simple if-else Statement

```
// Test for doubles
if (roll1 == roll2)
{
    System.out.println("Doubles");
}
else
{
    System.out.println("Sorry, no doubles");
}
// rest of program
```

Control Flow of if-else

```
if (roll1 == roll2)
{
    System.out.println("Doubles");
    // continue with rest of program
}
else
{
    System.out.println("Sorry, no doubles");
    System.out.println("Sorry ...");
    // continue with rest of program
```
Advanced Topic

- Selection Operator
  condition ? value1 : value2
  - If the condition is true, value1 is returned. value2 if the condition false

Ex. Absolute Value
  \[ y = (x >= 0) ? x : -x; \]

6.3 Multiple Alternatives

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";
}

Multiple alternatives

- First condition that is true is the ONLY statement executed
- Therefore, order matters
if (testScore >= 90) {
    System.out.println("Great!");
} else if (testScore >= 70) {
    System.out.println("OK");
} else if (testScore >= 50) {
    System.out.println("D’oh!");
}

Else matters also

if (testScore >= 90) {
    System.out.println("Great!");
} else if (testScore >= 70) {
    System.out.println("OK");
} else if (testScore >= 50) {
    System.out.println("D’oh!");
}

Not exclusive statements anymore

WRONG version

if (testScore >= 50) {
    System.out.println("D’oh!");
} else if (testScore >= 70) {
    System.out.println("OK");
} else if (testScore >= 90) {
    System.out.println("Great!");
}

Adv 6.2 Switch Statement

switch ( < arithmetic expression>){
    <case label 1>: <case body 1>
    ...
    <case label n>: <case body n>
}

- Can only be used on integers, characters, or enumerated constants
- Good for testing multiple values for one expression
6.3.2 Nested branches

- **then** and **else** blocks can contain as many statements as needed

- Can also have another if statement: nested-if statement

- The **break** statement causes execution to skip the remaining portion of the **switch** statement and resume execution following the **switch** statement.

- The **break** statement is necessary to execute statements in one and only one case.

Switch statement

```java
switch ( c )//Some primitive{
    case 'y': System.out.println("Yes");
             break;
    case 'n': System.out.println("No");
             break;
    default : System.out.println("Invalid entry");
}
```

Example

```java
if (testScore >= 70) {
    if (studentAge < 10) {
        System.out.println("You did a great job");
    } else {
        System.out.println("You did pass");
        //test score >=70 and age >=10
    }
} else { //test score < 70
    System.out.println("You did not pass");
}
```
Tax Schedule

Table 1  Federal Tax Rate Schedule (1992)

<table>
<thead>
<tr>
<th>Tax Bracket</th>
<th>Percentage</th>
<th>Tax Bracket</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2,000 ... $21,450</td>
<td>15%</td>
<td>$21,450 ...</td>
<td>15%</td>
</tr>
<tr>
<td>Amount over $21,450 up to $51,900</td>
<td>28%</td>
<td>Amount over $51,900 up to $86,500</td>
<td>28%</td>
</tr>
<tr>
<td>Amount over $86,500</td>
<td>31%</td>
<td></td>
<td>31%</td>
</tr>
</tbody>
</table>

Figure 6  Income Tax Computation Using 1992 Schedule

- 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

```java
if (status == SINGLE)
{
    if (income <= SINGLE_BRACKET1)
    {
        tax = RATE1 * income;
    }
    else if (income <= SINGLE_BRACKET2)
    {
        tax = RATE1 * SINGLE_BRACKET1 + RATE2 * (income - SINGLE_BRACKET1);
    }
    else
    {
        tax = RATE1 * SINGLE_BRACKET1 + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1) + RATE3 * (income - SINGLE_BRACKET2);
    }
}
```

else {
    if (income <= MARRIED_BRACKET1) {
        tax = RATE1 * income;
    } else if (income <= MARRIED_BRACKET2) {
        tax = RATE1 * MARRIED_BRACKET1 + RATE2 * (income - MARRIED_BRACKET1);
    } else {
        tax = RATE1 * MARRIED_BRACKET1 + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1) + RATE3 * (income - MARRIED_BRACKET2);
    }
}
```

Dangling else

- There is a good reason we always use curly braces in our programs
- Very common error can be made:
  ```java
  if (testScore >= 60)
    if(testScore <= 80)
      System.out.println("You are in safe range");
    else // Pitfall!
      System.out.println("You may be in trouble");
  ```
**Boolean Operators**

- What if we want one expression to test multiple conditions

- Ex. Do they have a score above 50 and below 75?

- Need *boolean operators*

**Truth table**

| P  | Q  | P&&Q | P||Q | !P |
|----|----|------|------|----|
| false | false | false | false | true |
| false | true | false | true | true |
| true | false | false | true | false |
| true | true | true | true | false |

**Boolean operators**

- Can use a logical operator to test simultaneously

  - `&&` AND true if both are true, false otherwise
  - `||` OR true if either is true, false otherwise
  - `!` NOT true of the expression is false, false if it is true

**Example**

```java
if (testScore > 50 && testScore < 75){
    System.out.print("In range C to D range");
}
```

```java
char c;
...
if (!('c == '!')){
    System.out.print(c);
}
```

**Many ways to write one expression:**

```java
if (age < 0){
    System.out.print("valid");
} else {
    System.out.print("invalid");
}
```

```java
if (!age >= 0){
    System.out.print("valid");
} else {
    System.out.print("invalid");
}
```

```java
if (age >= 0){
    System.out.print("invalid");
} else {
    System.out.print("valid");
}
```

**Evaluation**

- Like before, operations evaluated left to right

  Ex. – y = 0;
  
  ```
  x / y > z || y == 0  //Run time error, divide by zero
  y == 0 || x / y > z  //Legal
  ```
### Precedence Table

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Group</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>subexpression</td>
<td>( )</td>
<td>inner to outer left to right</td>
</tr>
<tr>
<td>8</td>
<td>unary</td>
<td>! ~</td>
<td>right to left</td>
</tr>
<tr>
<td>7</td>
<td>multiplicative</td>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>6</td>
<td>additive</td>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>5</td>
<td>relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>4</td>
<td>equality</td>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>3</td>
<td>AND</td>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>2</td>
<td>OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>assignment</td>
<td>=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

### Example

- In math, we represent the range of $x$
  - $80 \leq x < 90$

- How do we represent this as a boolean expression?
  - $80 \leq x \text{ and } x < 90$

### Common Uses

- Check validity of data
  - Divide by zero, out or range errors

- Classify data
  - Test above mean, below mean

- Flags
  - Store system settings or user preferences
  - Long messages, noise off, debug mode

### Style

- Equivalent statements give a boolean isRaining
  - `if (isRaining == true){}`
  - `if (isRaining){} //Preferred`

- Use good identifier names
  - `isMoving` vs. `motionStatus`
  - `isCheckedOut` vs. `bookStatus`

- **MOST COMMON ERROR!**
  - `if (x = 5){}`

### 6.2.3 Comparing Strings

- Boolean operators can be used on objects, but do they mean the same thing?

- To test two Strings, use equals() method

### String equality

- Don't use `==` for strings!
  - `if (input == "Y") // WRONG!!!`

- Use equals method:
  - `if (input.equals("Y"))`

  ```java
  string1.equals(string2);
  ```

- `==` tests identity, equals tests equal contents
equalsIgnoreCase()

- Case insensitive test ("Y" or "y")
  - if (input.equalsIgnoreCase("Y"))

  string1.equalsIgnoreCase(string2) // returns boolean

- compareTo returns which string comes before the other in the dictionary
- Returns an integer

6.2.4 Comparing Objects

- == tests for identity, equals for identical content
- Most classes have an equals() method defined
  - 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

6.2.4 Testing for null

- Reference variables store a reference (address) for an actual object
  - What do they store when they haven’t been set to refer to an object?

  - null is a Java reserved word to designate no object set

compareTo()

string1.compareTo(string2)

- Returns < 0 if string1 comes first
- Returns 0 if they are equal
- Returns > 0 if string 2 comes first

- "car" comes before "cargo"
- All uppercase letters come before lowercase: "Hello" comes before "car"

Figure 4 Comparing Object References

- 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 ""
Draw the Memory Diagram
String s1 = "Robby", s2;
s2 = s1;

Equality
What is the result of:
s1 == s2
because the variables store the same address

Draw the Memory Diagram
String s1 = "Robby", s2;
s2 = "Robby";

Equality
What is the result of:
str1 == str2
because the variables store the same address

Draw the Memory Diagram
String s1 = "Robby", s2;
s2 = new String( s1 );

Equality
What is the result of:
s1 == s2
Equality

- What is the result of:
  
  \[ \text{s1} == \text{s2} \]

- How do you compare the contents?
  
  \[ \text{s1}.equals(\text{s2}) \]

- Why?

    because the variables store different addresses

Equality

- How do you compare the contents?
  
  \[ \text{s1}.equals(\text{s2}) \]

- Why?

    because now the contents are compared by the equals method

String

- Draw a memory diagram for:
  
  String \text{s1} = "Jimmy";
  String \text{s2};
  \text{s2} = \text{s1};
  \text{s1} = "Bobby";

String Concatenation

- The + operator
- Draw a memory diagram for:
  
  String \text{s1} = "Jimmy";
  \text{s1} = \text{s1} + "Bobby";

6.4 Using Boolean Expressions

- 6.4.1 introduces boolean data type
- Pioneered by George Boole

double \text{temp} = 100;
\text{System.out.println}("Is it hot?: " + (\text{temp} > 60));
6.4.2 Predicate methods

- A predicate method returns a boolean value
  ```java
  public boolean isOverdrawn() {
    return balance < 0;
  }
  ```
- Use to test conditions – just like another other method
  ```java
  if (harrysChecking.isOverdrawn()) ... 
  ```

Example

- Scanner class has a `hasNextInt()` method which allows you to test if there is an int before you get it (and cause program to crash)
  ```java
  Scanner in = new Scanner(System.in)
  int score;
  System.out.println("Enter Score");
  if(in.hasNextInt())
    score = in.nextInt()
  ```

Better Solution

- Create methods in our class to handle this problem
- String method uses `equals()` method to compare two strings
  ```java
  public class Weight{
      private int gram;
      ...
      public boolean equals(Weight wgt) {
        boolean result;
        if (this.gram == wgt.gram){
          result = true;
        } else {
          result = false;
        }
        return result;
      }
      ...
  }
  ```