Chapter 3

Flow of Control

- Branching
- Loops
- exit(n) method
- Boolean data type and expressions
What is “Flow of Control”? 

- Flow of Control is the execution order of instructions in a program
- All programs can be written with three control flow elements:
  1. **Sequence** - just go to the next instruction
  2. **Branching** or **Selection** - a choice of at least two
     - either go to the next instruction
     - or jump to some other instruction
  3. **Loop** or **Repetition** - a loop (repeat a block of code)
     at the end of the loop
     - either go back and repeat the block of code
     - or continue with the next instruction after the block
Java Flow Control Statements

Sequence

- the default
- Java automatically executes the next instruction unless you use a branching statement

Branching

- if
- if-else
- if-else if-else if- … - else
- switch

Loop

- while
- do-while
- for
Java `if` Statement

- Simple selection
- Do the next statement if test is true or skip it if false
- Syntax:

  ```java
  if (Boolean_Expression)
      Action if true;  //execute only if true
  next action;  //always executed
  ```

- Note the indentation for *readability* (not compile or execution correctness)
if Example

if(eggsPerBasket < 12)
    //begin body of the if statement
    System.out.println("Less than a dozen eggs per basket");
    //end body of the if statement
totalEggs = numberOfEggs * eggsPerBasket;
System.out.println("You have a total of
        + totalEggs + " eggs.");

- The body of the if statement is conditionally executed
- Statements after the body of the if statement always execute
Multiple Statements

- **Action if true** can be either a single Java statement or a set of statements enclosed in braces `{ }`.
- A set of statements in braces is called a **compound statement** and can be used anywhere a single statement can be used.

```java
if(eggsPerBasket < 12) {
    //begin body of the if statement
    System.out.println("Less than a dozen ...");
    costPerBasket = 1.1 * costPerBasket
} //end body of the if statement

totalEggs = numberOfEggs * eggsPerBasket;
System.out.println("You have a total of "+ totalEggs + " eggs.");
```
Two-way Selection: **if-else**

- Select either one of two options
- Either do Action1 or Action2, depending on test value
- Syntax:

```java
if (Boolean_Expression)
{
    Action1 //execute only if true
}
else
{
    Action2//execute only if false
}
Action3//always executed
```
**if-else Examples**

- **Example with single-statement blocks:**
  ```java
  if(time < limit)
      System.out.println("You made it.");
  else
      System.out.println("You missed the deadline.");
  ```

- **Example with compound statements:**
  ```java
  if(time < limit)
      {
          System.out.println("You made it.");
          bonus = 100;
      }
  else
      {
          System.out.println("You missed the deadline.");
          bonus = 0;
      }
  ```
Definition of Boolean Values

- Branching: there is more than one choice for the next instruction
- Which branch is taken depends on a test condition which evaluates to either true or false
- In general:
  if test is true then do this, otherwise it is false, do something else
- Variables (or expressions) that are either true or false are called boolean variables (or expressions)
- So the value of a boolean variable (or expression) is either true or false
- boolean is a primitive data type in Java
Boolean Expressions

- Boolean expressions can be thought of as test conditions (questions) that are either true or false.
- Often two values are compared.
- For example:
  Is A greater than B?
  Is A equal to B?
  Is A less than or equal to B?
  etc.
- A and B can be any data type (or class), but they should be the same data type (or class).
Java Comparison Operators

<table>
<thead>
<tr>
<th>Math Notation</th>
<th>Name</th>
<th>Java Notation</th>
<th>Java Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
<td>==</td>
<td>balance == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer = 'y'</td>
</tr>
<tr>
<td>≠</td>
<td>not equal to</td>
<td>!=</td>
<td>income ≠ tax</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer != 'y'</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>&gt;</td>
<td>income &gt; outgo</td>
</tr>
<tr>
<td>≥</td>
<td>greater than or equal to</td>
<td>&gt;=</td>
<td>points &gt;= 60</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>&lt;</td>
<td>pressure &lt; max</td>
</tr>
<tr>
<td>≤</td>
<td>less than or equal to</td>
<td>&lt;=</td>
<td>income &lt;= outgo</td>
</tr>
</tbody>
</table>
Compound Boolean Expressions

• Use `&&` to AND two or more conditions
  » Expression will be true if both parts are true.
  » \( A \&\& B \) will be true if both \( A \) and \( B \) are true

• Use `||` to OR two or more conditions
  » Expression will be true if either part is true, or if both parts are true.
  » \( A \mid\mid B \) will be true if either \( A \) or \( B \) is true, or if both \( A \) and \( B \) are true.
Compound Boolean Expressions

- Example: write a test to see if B is either 0 or between the values of B and C

\[(B == 0) || (A <= B && B < C)\]

- In this example the parentheses are not required but are added for clarity
  - See text (and later slides) for Precedence rules
  - Note the short-circuit, or lazy, evaluation rules in text (and later in slides)
  - Use a single & for AND and a single | for OR to avoid short-circuit evaluation and force complete evaluation of a boolean expression
Java Comparison Methods for String Class

- "==" does not do what you may think for String objects
  - When "==" is used to test objects (such as String objects) it tests to see if the storage *addresses* of the two objects are the same
    - are they stored in the same *location*?
    - more will be said about this later
- Use ".equals" method to test if the strings, themselves, are equal
  ```java
  String s1 = "Mondo";
  String s2;
  s2 = SavitchIn.readLine();
  //s1.equals(s2) returns true if the user enters Mondo, false otherwise
  ```
- ".equals() is case sensitive"
- Use ".equalsIgnoreCase() to ignore case"
Alphabetical Ordering

- Use `compareTo` method of `String` class
- Uses ASCII lexicographic ordering where all uppercase letters come before all lowercase letters
  - For example capital 'Z' comes before small 'a'
  - Convert strings to all uppercase (or all lowercase) to avoid problems
- `s1.compareTo(s2)`
  - returns a negative value if `s1` comes before `s2`
  - returns zero if the two strings are equal
  - returns a positive value if `s2` comes before `s1`

```java
// Assume s1 and s2 are two string variables
// that have been given string values.
String upperS1 = s1.toUpperCase();
String upperS2 = s2.toUpperCase();
if (upperS1.compareTo(upperS2) < 0)
    System.out.println(s1 + " precedes " + s2);
```
Nested if Statements

- One if statement can have another if statement inside it.
- These are called *nested if statements*.
- Inner statements are indented more than outer statements.

```java
if (balance >= 0)
    if (RATE >= 0)
        balance = balance + (RATE * balance)/12;
    else
        System.out.println("Cannot have negative rate");
else
    balance = balance – OVERDRAWN_PENALTY;
```

The inner statement will be skipped entirely if `balance >= 0` is false.
Multibranch selection: if-else if-else if-else

- One way to handle situations with more than two possibilities
- Syntax:

  
  ```java
  if(Boolean_Expression_1)
      Action_1
  else if(Boolean_Expression_2)
      Action_2
      .
      .
      .
  else if(Boolean_Expression_n)
      Action_n
  else
      Default_Action
  ```
if-else if-else if-...-else

Example

```java
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'E';
```

Note indentation.

Even though these are nested if statements, they are all indented the same amount to indicate a multibranch selection.
Multibranch selection: `switch`

```java
switch (Controlling_Expression) {
    case Case_Label:
        statements
        ...
        break;
    case Case_Label:
        statements
        ...
        break;
    default:
        statements
        ...
        break;
}
```

- Another way to program multibranch selection
- Uses `Controlling_Expression` to decide which way to branch
  - `Controlling_Expression` must be `char, int, short` or `byte`.
  - `Controlling Expression` and `Case_Label` must be same type.
Multibranch selection: `switch`

```
switch (Controlling_Expression) {
    case Case_Label:
        statements
        ...
        break;
    case Case_Label:
        statements
        ...
        break;
    default:
        statements
        ...
        break;
}
```

- **When a `break` statement is encountered**, control goes to the first statement after the `switch`.
- `break` **may be omitted**
- Can have **any number of cases**
- **Default case** is optional
switch Example

```java
switch (seatLocationCode) {
    case 1:
        System.out.println("Orchestra");
        price = 40.00;
        break;
    case 2:
        System.out.println("Mezzanine");
        price = 30.00;
        break;
    case 3:
        System.out.println("Balcony");
        price = 15.00;
        break;
    default:
        System.out.println("Unknown seat code");
        break;
}
```

Output if `seatLocationCode` is 2:

Mezzanine
Repetition: Loops

• Structure:
  » Usually some initialization code
  » body of loop
  » loop termination condition

• Several logical organizations
  » counting loops
  » sentinel-controlled loops
  » infinite loops
  » minimum of zero or minimum of one iteration

• Several programming statement variations
  » while
  » do-while
  » for
**while Loop**

- **Syntax:**
  
  ```java
  while(Boolean_Expression)
  {
    //body of loop
    First_Statement;
    ...
    Last_Statement;
  }
  ```

  Something in body of loop should eventually cause `Boolean_Expression` to be *false*.

- Initialization statements usually precede the loop.
- `Boolean_Expression` is the loop termination condition.
- The loop will continue executing as long as `Boolean_Expression` is true.
- May be either counting or sentinel loop
  - Good choice for sentinel loop
Semantics of the **while** Statement

```
while (Boolean_Expression)
    Body
```

Start

Evaluate **Boolean_Expression**

- **true**
  - Execute **Body**
- **false**
  - End loop
**while**: A Counting Loop Example

- A loop to sum 10 numbers entered by user

```java
int next;
//Loop initialization
int count = 1;
int total = 0;
while(count <= 10) //Loop termination condition
{
    //Body of loop
    next = SavitchIn.readLineInt();
    total = total + next;
    count++;
    //Loop termination counter
}
```
while:
A Sentinel Controlled Loop Example

- A loop to sum positive integers entered by the user
- `next` is the sentinel
- The loop terminates when the user enters a negative number

```java
//Initialization
int next = 0;
int total = 0;
while(next >= 0) //Termination condition
{
    //Body
    total = total + next;
    next = SavitchIn.readLineInt();
}
```
while: A Minimum of Zero Iterations

- Because the first input value read and the test precedes the loop, the body the while loop body may not execute at all

```java
//Initialization
int next;
int total = 0;
next = SavitchIn.readLineInt();
while(next >= 0)//Termination condition
{ //Body
    total = total + next;
    next = SavitchIn.readLineInt();
}
```

- If the first number the user enters is negative the loop body never executes
**do-while Loop**

- **Syntax**

  ```java
do
  {  //body of loop
      First_Statement;
      ...
      Last_Statement;
  }
  while(Boolean_Expression);
```

- Initialization code may precede loop body
- Loop test is after loop body so the body must execute at least once (minimum of at least one iteration)
- May be either counting or sentinel loop
  - Good choice for sentinel loop

Something in body of loop should eventually cause `Boolean_Expression` to be `false`. 
Semantics of the **do-while** Statement

```
do
    Body
while (Boolean_Expression);
```

**Start**

- **Execute Body**

- **Evaluate Boolean_Expression**
  - **true** → **Execute Body**
  - **false** → **End loop**
do-while Example

```java
int count = 1;
int number = 5;
do  //Display integers 1 to 5 on one line
{  
    System.out.print(count + " ");
    count++;
}
while(count <= number);
```

Note that `System.out.print()` is used and not `System.out.println()` so the numbers will all be on one line.

Output:

```
1 2 3 4 5
```
for Loop

- Good choice for counting loop
- Initialization, loop test, and loop counter change are part of the syntax
- Syntax:

  ```java
  for(Initialization; Boolean_Expression; Update_Action)
  loop body;
  ```
Semantics of the **for** Statement

```
for (Initialization; Boolean_Expression; Update_Action)
  loop body;
```

**Diagram:**
- Start
  - Evaluate **Boolean_Expression**
    - true
      - Execute **Body**
    - false
      - End loop
  - Execute **Initialization**
  - Execute **Update_Action**
for Example

- Count down from 3 to 1

```java
for(int count = 3; count >= 1; count--)
{
    System.out.print("T = " + count);
    System.out.println(" and counting");
}
System.out.println("Blast off!");
```

Output:

```
T = 3 and counting
T = 2 and counting
T = 1 and counting
Blast off!
```
The `exit` Method

- If you have a program situation where it is pointless to continue execution you can terminate the program with the `exit(n)` method.
- \( n \) is often used to identify if the program ended normally or abnormally.
- \( n \) is *conventionally* 0 for normal termination and non-zero for abnormal termination.
System.out.println("Enter e to exit or c to continue");
char userIn = SavitchInReadLineChar();
if(userIn == 'e')
    System.exit(0);
else if(userIn == 'c')
{
    //statements to do work
}
else
{
    System.out.println("Invalid entry");
    //statements to something appropriate
}
Nested Loops

- The body of a loop can have any kind of statements, including another loop.

```java
for (line = 0; line < 4; line++)
    for (star = 0; star < 5; star++)
        System.out.print('*');
    System.out.println();
```

- Each time the outer loop body is executed, the inner loop body will execute 5 times, making a total of 20 times.

Output:

```
*****
*****
*****
*****
*****
```
Some Practical Considerations When Using Loops

- The most common loop errors are unintended infinite loops and off-by-one errors in counting loops.
- Sooner or later everyone writes an unintentional infinite loop
  - To get out of an unintentional infinite loop enter ^C (control-C)
- Loops should be tested thoroughly, especially at the boundaries of the loop test, to check for off-by-one and other possible errors.
Tracing a Variable in a Loop

- *Tracing a variable*: print out the variable each time through the loop

- A common technique is to test loop counters and troubleshoot off-by-one and other loop errors.

- Some systems provide a built-in tracing system that allows you to trace a variable without having to change your program.

- If no built-in utility is available, insert temporary output statements to print values.
The Type boolean

- A primitive type
- Can have expressions, values, constants, and variables just as with any other primitive type
- Only two values: true and false
- Can use a boolean variable as the condition in an if statement

```
if (systemsAreOK)
    System.out.println("Initiate launch sequence.");
else
    System.out.println("Abort launching sequence");
```

- Using a boolean variable as the condition can make an if statement easier to read by avoiding a complicated expression.
**boolean Variables in Assignments**

- A boolean expression evaluates to one of the two values true or false.
- The value of a boolean expression can be assigned to a boolean variable:

```java
int number = -5;
boolean isPositive;
isPositive = (number > 0);
if (isPositive)
    System.out.println("positive");
else
    System.out.println("negative or zero");
```

- There are simpler and easier ways to write this small program, but boolean variables are useful in keeping track of conditions that depend on a number of factors.
Truth Tables for **boolean** Operators

<table>
<thead>
<tr>
<th>Value of A</th>
<th>Value of 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>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

| Value of A | Value of B | A || B |
|------------|------------|-------|
| true       | true       | true  |
| true       | false      | true  |
| false      | true       | true  |
| false      | false      | false |

<table>
<thead>
<tr>
<th>Value of 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>
Precedence

An example of using precedence rules to see which operators in following expression should be done first:

\[ \text{score} < \frac{\text{min}}{2} - 10 \ || \ \text{score} > 90 \]

- Division operator has highest precedence of all operators used here so treat it as if it were parenthesized:

\[ \text{score} < \left(\frac{\text{min}}{2}\right) - 10 \ || \ \text{score} > 90 \]

- Subtraction operator has next highest precedence:

\[ \text{score} < \left(\frac{\text{min}}{2} - 10\right) \ || \ \text{score} > 90 \]

- The < and > operators have equal precedence and are done in left-to-right order:

\[ (\text{score} < \left(\frac{\text{min}}{2} - 10\right)) \ || \ (\text{score} > 90) \]

- The last expression is a fully parenthesized expression that is equivalent to the original. It shows the order in which the operators in the original will be evaluated.
Precedence Rules

**Highest Precedence**
- First: the unary operators: `+`, `-`, `++`, `--`, and `!`
- Second: the binary arithmetic operators: `*`, `/`, `%`
- Third: the binary arithmetic operators: `+`, `-`
- Fourth: the boolean operators: `<`, `>`, `=<`, `>=`
- Fifth: the boolean operators: `==`, `!=`
- Sixth: the boolean operator `&`
- Seventh: the boolean operator `|`
- Eighth: the boolean operator `&&`
- Ninth: the boolean operator `||`

**Lowest Precedence**
Short-Circuit Evaluation

- *Short-circuit evaluation*—only evaluating as much of a boolean expression as necessary.

- Example:

  ```java
  if ((assign > 0) && ((total/assign) > 60))
      System.out.println(“Good work”);
  else
      System.out.println(“Work harder.”);
  ```

- If `assign > 0` is false, then the complete expression cannot be true because AND is only true if both operands are true.

- Java will not evaluate the second part of the expression.

- Short-circuit evaluation prevents a divide-by-zero exception when `assign` is 0.
Summary
Part 1

- Java selection statements: *if, if-else, if-else if, and switch*
- Java repetition (loop) statements: *while, do-while, and for*
- Loops can be counter or sentinel controlled
- Any loop can be written any of the three loop statements, but
  - *while and do-while are good choices for sentinel loops*
  - *for is a good choice for counting loops*
Summary
Part 2

- Unintended infinite loops can be terminated by entering `^C` (control-C)
- The most common loop errors are unintended infinite loops and off-by-one errors in counting loops
- Branching and loops are controlled by `boolean` expressions
  - `boolean` expressions are either `true` or `false`
  - `boolean` is a primitive data type in Java
- `exit(n)` is a method that terminates a program
  - `n = 0` is the conventional value for normal termination