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. **Selection** - a choice of at least two
     – either go to the next instruction
     – or jump to some other instruction
  3. **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
- Selection and Repetition are called *Branching* since these are branch points in the flow of control

Java Flow Control Statements

Sequence

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

Branching: Selection

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

Branching: Repetition

- while
- do-while
- for

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

Math Notation	Name	Java Notation	Java Examples
=	equal to	==	balance == 0 answer = 'y'
≠	not equal to	!=	income ≠ tax answer != 'y'
>	greater than	>	income > outgo
≥	greater than or equal to	>=	points >= 60
<	less than	<	pressure < max
≤	less than or equal to	<=	income <= outgo

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

```
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

Compound Boolean Expressions

- Use && to AND two or more conditions
- Use `||` to OR two or more conditions
- See text for definitions of AND and OR
- For 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 **if** statement

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

```
 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

Java Statement Blocks: Compound Statements

- `Action if true` can be either a single Java statement or a set of statements enclosed in curly brackets (a *compound* statement, or *block*)
- For example:

All statements between braces are controlled by `if`

```
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:

```
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:

```
if(time < limit)
   System.out.println(“You made it.”);
else
   System.out.println(“You missed the deadline.”);
```

- Example with compound statements:

```
if(time < limit)
{
   System.out.println(“You made it.”);
   bonus = 100;
}
else
{
   System.out.println(“You missed the deadline.”);
   bonus = 0;
}
```

Multibranch selection: **if-else if-else if-…-else**

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

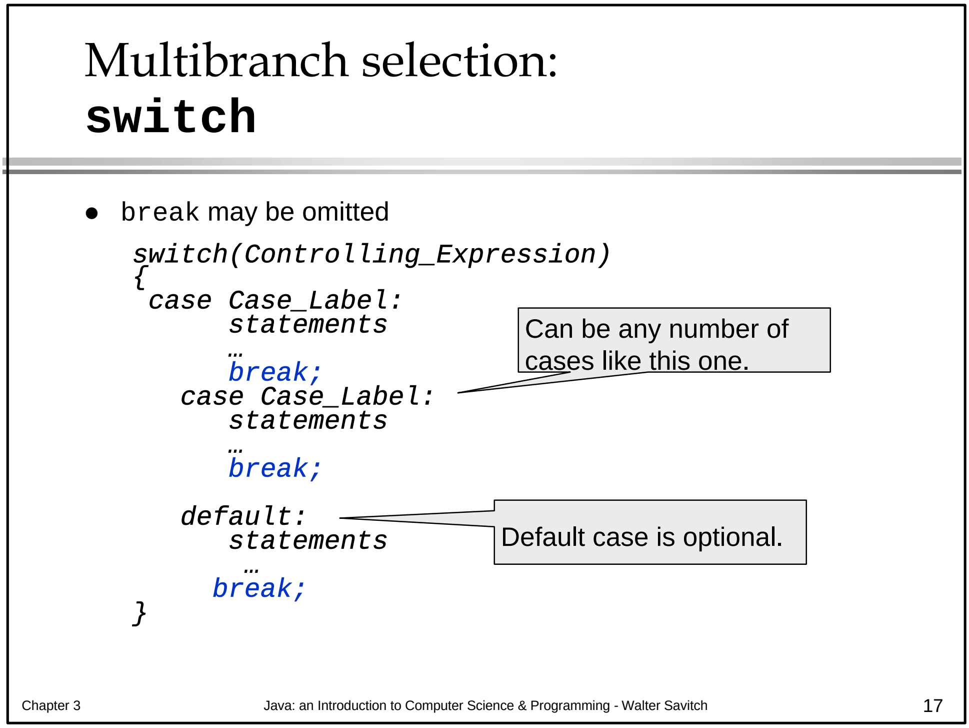
```
if(Boolean_Expression_1)
   Action_1
else if(Boolean_Expression_2)
   Action_2
      .
      .
      .
else if(Boolean_Expression_n)
   Action_n
else
   Default_Action
```

`if-else if-else if-…-else` Example

```
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’;
```

Multibranch selection: **`switch`**

- Another way to program multibranch selection
- *`Controlling_Expression`* must be *`char, int, short`* *or* *`byte`*
- *`Controlling Expression`* and *`Case_Label`* must be same type
- When a `break` statement is encountered, control goes to the first statement after the `switch`.



Multibranch selection: `switch`

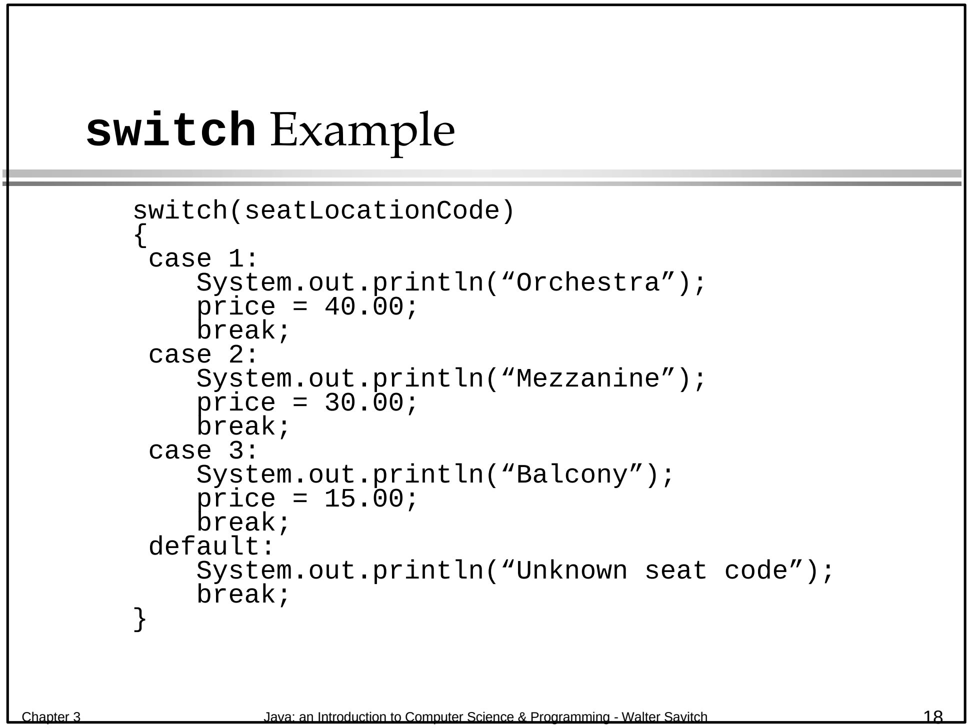
- `break` may be omitted

```
switch(Controlling_Expression)
{
 case Case_Label:
      statements
      …
      break;
   case Case_Label:
      statements
      …
      break;

   default:
      statements
       …
     break;
}
```

Can be any number of cases like this one.

Default case is optional.



`switch` Example

```
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;
}
```

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:

```
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.
- May be either counting or sentinel loop
  - » Good choice for sentinel loop

`while :` a counting loop example

- A loop to sum 10 numbers entered by user

```
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

```
//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

```
//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

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

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

- 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

do-while Example

```
int count = 1;
int number = 10;
do //Display integers 1 - 10 on one line
{
   System.out.print(count + “, “ );
   count++;
}while(count <= number);
```

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

for Loop

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

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

`for` Loop

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

- Execution sequence:
  1. *`Initialization`* - ***<u>executes only once, before the loop body is executed the first time</u>***
- 2. *`Boolean_Expression`* - the loop test
  3. *`loop body`* - execute only if loop test is *`true`*
  4. *`After_Loop_Body`* - typically changes the loop counter
  5. *`Boolean_Expression`* - ***<u>Repeat the loop test (step 2), etc.</u>***

`for` Example

- Count down from 9 to 0

```
for(int count = 9; count >= 0; count--)
{
   System.out.print("T = " + count);
   System.out.println(" and counting");
}
System.out.println("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

exit Method Example

```
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
}
```

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 unintended infinite loop enter `^C` (control-C)
- Loops should 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 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:

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

Parentheses are not necessary here.

Parentheses are necessary here.

- 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

&& (and)

Value of A	Value of B	A && B
true	true	**true**
true	false	**false**
false	true	**false**
false	false	**false**

|| (or)

Value of A	Value of B	A || B
true	true	**true**
true	false	**true**
false	true	**true**
false	false	**false**

! (not)

Value of A	!A
true	**false**
false	**true**

Precedence

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

```
score < min/2 – 10 || score > 90
```

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

```
score < (min/2) – 10 || score > 90
```

- Subtraction operator has next highest precedence :

```
score < ((min/2) – 10) || score > 90
```

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

```
(score < ((min/2) – 10)) || (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:

```
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