Chapter 5

Flow of Control Part 1:
Selection

Topics

- Forming Conditions
- *if/else* Statements
- Comparing Floating-Point Numbers
- Comparing Objects
  - The *equals* Method
  - *String* Comparison Methods
- The Conditional Operator (?:)
- The *switch* Statement

Flow of Control

- Sequential
  - Execute instructions in order
- Method calls
  - Transfer control to method, execute instructions in method, then return with or without a value
- Selection
  - Execute different instructions depending on data
- Looping
  - Repeat a set of instructions for different data

Equality Operators

- Used to determine if values of two expressions are equal or not equal
- Result is *true* or *false*

Equality operators	Type (number of operands)	Meaning
= =	binary	is equal to
!=	binary	is not equal to

Examples

- If *int* variable *age* holds the value 32:

  `( age == 32 )` evaluates to *true*

  `( age != 32 )` evaluates to *false*

Use the equality operators only with primitive types and object references, not to compare object data!

- Do not confuse the equality operator (==) with the assignment operator (=).

Relational Operators

- Used to compare the values of two expressions
- Result is *true* or *false*

Relational Operators	Type (number of operands)	Meaning
<	binary	is less than
<=	binary	is less than or equal to
>	binary	is greater than
>=	binary	is greater than or equal to

Example

- If *int* variable *age* holds value 32:

`( age < 32 )` evaluates to *false*

`( age <= 32 )` evaluates to *true*

`( age > 32 )` evaluates to *false*

`( age >= 32 )` evaluates to *true*

Logical Operators

Logical Operator	Type (number of operands)	Meaning
!	Unary	NOT
&&	Binary	AND
||	Binary	OR

Operands must be boolean expressions!

Logical Operators

- The NOT operator (!) inverts the value of its operand. If the operand is *true*, the result will be *false*; and if the operand is *false*, the result will be *true*.
- The AND operator (&&) takes two *boolean* expressions as operands; if both operands are *true,* the result will be *true*, otherwise it will be *false*.
- The OR operator (||) takes two *boolean* expressions as operands. If both operands are *false*, the result will be *false*; otherwise it will be *true*.

Truth Table

a	b	!a	a && b	a || b
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

For operator precedence, see Appendix B

Short-Circuit Evaluation

- For any logical operator, the operands are evaluated left to right
- If the result of the logical operation can be determined after evaluating the first operand, the second operand is not evaluated.
  - If the first operand of an `||` is *true*, the result will be *true*
  - If the first operand of an `&&` is *false*, the result will be *false*
- *See Example 5.1 Logical Operators.java*

Suppose we have three *ints x, y,* and *z,* and we want to test if *x* is less than both *y* and *z*. A common error is to express the condition this incorrect way:

```
x < y && z  // compiler error
```

Each operand of a logical operator must be a boolean expression. This is correct:

```
x < y  &&  x < z
```

Equivalence of Expressions

DeMorgan's Laws:

1. NOT(A AND B) = (NOT A) OR (NOT B)
2. NOT(A OR B) = (NOT A) AND (NOT B)

- Thus to find an equivalent expression:
  - change `&&` to `||`
  - change `||` to `&&`
  - negate each operand expression

Negation of Equality and Relational Operators

Expression	!(Expression)
`a == b`	`a != b`
`a != b`	`a == b`
`a <  b`	`a >= b`
`a <= b`	`a > b`
`a > b`	`a <= b`
`a >= b`	`a  < b`

Examples

These expressions are equivalent:

```
( age <= 18 || age >= 65 )

!( age > 18 && age < 65 )

!( age > 18 ) || !( age < 65 )
```

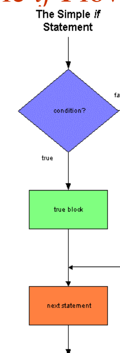
Simple *if* Statement

- Used when program should perform an operation for one set of data, but do nothing for all other data
- Syntax:

```
if ( condition )
{
   // true block
   // executed if condition is true
}
```

- Curly braces are optional if true block contains only one statement

Simple *if* Flow of Control



- Indent the true block of the *if* statement for clarity
- Line up the open and closing curly braces under the "i" in *if*

Simple *if* Example

- *See Example 5.2 PassingGrade.java*

```
public class PasingGrade {
publinc static main(String [ ]args) {
if (grade >= 60)
    System.out (“Pass”);
System.out (“No Good”);
}
}
```

Do not put a semicolon after the condition. Doing so indicates that the true block is empty and can cause a logic error at run time.

if /else

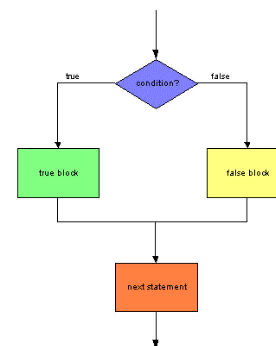
- Used when data falls into two mutually exclusive categories and program should perform different operations for each set
- Sample uses:
  - If password is correct, welcome user; otherwise, ask for reentry.
  - If person is old enough to vote, issue a voting card; otherwise, refuse the request.

if/else Syntax

```
if ( condition )
{
    // true block
}
else
{
    // false block
}
```

- Again, curly braces are optional for either block that consists of only one statement
- Note indentation of true and false blocks for readability

if/else Flow of Control



Example

- *See Example 5.3 Divider.java*

```
public class PasingGrade {
publinc static main(String [ ]args) {
if (grade >= 60)
    System.out ("Pass");
else
    System.out ("No Good");
}
}
```

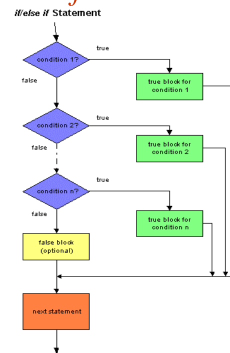
if/else if

- Used when data falls into multiple mutually exclusive categories and program should do different operations for each set
- Ex:
  - Determine letter grade based on numeric grade
  - Determine ticket price (different prices for child, adult, and senior)

if/else if Syntax

```
if ( condition 1 )
{
    // true block for condition 1
}
else if (condition 2 )
{
    // true block for condition 2
}
…
else
   // false block for all conditions
```

if/else if Flow of Control



if/else if Example

- *See Example 5.4 LetterGrade.java*

Finding the Smallest of Three Numbers

```
read number1
read number2
read number3

if number1 is less than number2
    smallest is number1
else
    smallest is number2

if number3 is less than smallest
    smallest is number3
```

See Example 5.5 FindSmallest.java

Nested *if* Statements

- *if* statements can be written as part of the true or false block of another *if* statement.
- Typically, you nest *if* statements when more information is required beyond the results of the first *if* condition
- The compiler matches any *else* clause with the most previous *if* statement that doesn't already have an *else* clause.
- You can use curly braces to force a desired *if/else* pairing.

Example

```
if ( x == 2 )
  if ( y == x )
    System.out.println( "x and y equal
  2" );
  else
    System.out.println( "x equals 2,"
                        + " but y does
  not" );
```

- The *else* clause is paired with the second *if* , that is: `if ( y == x )`

Another Example

```
if ( x == 2 )
{
  if ( y == x )
     System.out.println( "x and y equal
  2" );
}
else
  System.out.println( "x does not equal
  2" );
```

- With curly braces added, the *else* clause is paired with the first *if* , that is: `if ( x == 2 )`

The "Dangling *else*"

- A **dangling *else*** is an *else* clause that cannot be paired with an *if* condition

```
if ( x == 2 )
  if ( y == x )
     System.out.println( "x and y equal 2" );
  else // paired with ( y == x )
     System.out.println( "y does not equal 2"
  );
else  // paired with ( x == 2 )
  System.out.println( "x does not equal 2" );

else  // no matching if!
  System.out.println( "x and y are not equal"
  );
```

- Generates the compiler error: 'else' without 'if'

Example 5.6: Generate a Secret Number

```
generate a secret random number between 1 and 10
prompt the user for a guess

if guess is not between 1 and 10
  print message
else
  if guess equals the secret number
     print congratulations
  else
     print the secret number
     if ( guess is within 3 numbers )
         print "You were close"
     else
         print "You missed by a mile"
     print "Better luck next time"
```

Testing Techniques

- Execution Path Testing
  - Develop a **test plan** that includes running the program multiple times with data values that cause all true and false blocks to be executed.
  - Check results against the program specifications
- Black Box Testing
  - Treat program like a black box (we don't know how the code is written)
  - Develop test data based on program specifications

When testing your program, develop input values that execute all possible paths and verify that the logic correctly implements the program specifications.

Comparing Floating-Point Numbers

- With IEEE 754 floating-point representation, minor rounding errors can occur in calculations
- *See Example 5.8*. We compute 11 * .1 two ways
  1. Multiplying 11 * .1, the result is 1.1
  2. Adding .1 11 times, the result is 1.0999999…
- These values will not compare as equal using the equality operator (==)
- We get similar results when assigning the same value to a *float* variable and to a *double* variable, then comparing the values.

Solution

- Choose a small **threshold value --** how close should the values be to be considered equal?
- If the difference between the two values is less than the threshold value, then we will consider the two floating-point numbers to be equal.
- Hint: use the *Math.abs* method to compute the difference.
- *See Example 5.9 ComparingFloatingPoint.java*

Comparing Objects

- The equality operator (`==`) compares object references.
- Example:
  - If *d1* and *d2* are two *Date* object references, then

    `( d1 == d2 )`

    evaluates to *true* only if *d1* and *d2* point to the same object, that is, the same memory location.

  *** The equality operator does not compare the data (*month, day,* and *year*) in those objects.

Comparing Object Data

- To compare object data, use the *equals* method

Return type	Method name and argument list
`boolean`	`equals( Object obj )` returns true if the data of the object *obj* is equal to the data in the object used to call the method

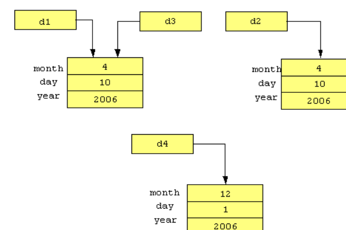
- Example (with *d1* and *d2 Date* object references):

  `d1.equals( d2 )`

  returns *true* if the *month, day*, and *year* of *d1* equals the *month, day,* and *year* of *d2*.

Comparing *Date* Objects

- See *Example 5.10 ComparingObjects.java*



- Do not use the equality operators (`==`, `!=`) to compare object data; instead, use the *equals* method.

Comparing *Strings*

- *Strings* are objects
- Thus to compare two *Strings*, use the *equals* method
- Example: *s1* and *s2* are *Strings*

  `s1.equals( s2 )`

  returns *true* only if each character in *s1* matches the corresponding character in *s2*
- Two other methods of the *String* class also can be used for comparing *Strings*:

  equalsIgnoreCase

  compareTo

The *equalsIgnoreCase* Method

Return type	Method name and argument list
`boolean`	`equalsIgnoreCase( String str )` compares the value of two *Strings*, treating uppercase and lowercase characters as equal. Returns *true* if the *Strings* are equal; returns *false* otherwise.

- Example:

```
String s1 = "Exit", s2 = "exit";
if ( s1.equalsIgnoreCase( s2 ) )
   System.exit( 0 );
```

The *compareTo* Method

Return type	Method name and argument list
`int`	`compareTo( String str )` compares the value of the two *Strings*. If the *String* object is less than the *String* argument, *str*, a negative integer is returned. If the *String* object is greater than the *String* argument, a positive number is returned; if the two *Strings* are equal, 0 is returned.

- A character with a lower Unicode numeric value is considered less than a character with a higher Unicode numeric value.
- *a* is less than b and *A* is less than *a*
- *See Example 5.11 ComparingStrings.java*

The Conditional Operator (?:)

- The conditional operator (?:) contributes one of two values to an expression based on the value of the condition.
- Some uses are
  - handling invalid input
  - outputting similar messages.
- Syntax:

  `( condition ? trueExp : falseExp )`

  If *condition* is *true*, *trueExp* is used in the expression

  If *condition* is *false*, *falseExp* is used in the expression

Equivalent Code

- The following statement stores the absolute value of the integer *a* into the integer *absValue*.

```
int absValue = ( a > 0 ? a : -a );
```

- The equivalent statements using *if/else* are:

```
int absValue;
if ( a > 0 )
   absValue = a;
else
   absValue = -a;
```

See Example 5.12 DoorPrize.java

See Appendix B Operator Precedence

The *switch* Statement

- Sometimes the *switch* statement can be used instead of an *if/else/if* statement for selection.
- Requirements:
  - we must be comparing the value of a character (*char*) or integer (*byte, short,* or *int*) expression to constants of the same types

Syntax of *switch*

```
switch ( char or integer expression )
{
   case constant1:
        // statement(s);
       break;     // optional
   case constant2:
       // statement(s);
       break;    // optional
       …
   default:     // optional
       statement(s);
        …
}
```

Operation of *switch*

- The expression is evaluated, then its value is compared to the *case* constants in order.
- When a match is found, the statements under that *case* constant are executed in sequence until either a ***break*** statement or the end of the ***switch*** block is reached.
- Once a match is found, if other *case* constants are encountered before a *break* statement, then the statements for these *case* constants are also executed.

Some Finer Points of *switch*

- The ***break*** statements are optional. Their job is to terminate execution of the ***switch*** statement.
- The ***default*** label and its statements, are also optional. They are executed when the value of the expression does not match any of the *case* constants.
- The statements under the *case* constant are also optional, so multiple *case* constants can be written in sequence if identical operations will be performed for those values.

Example: a Simple Calculator

- Prompt user for two *doubles* (*num1*, *num2*) and a *char* (*operation),* which can be '*a*' for addition or '*s*' for subtraction

```
switch ( operation )
{
  case 'a':
     result = num1 + num2;
     break;
  case 's':
     result = num1 - num2;
     break;
}
```

A Case-Insensitive Calculator

```
switch ( operation )
{
  case 'a':
  case 'A':
     result = num1 + num2;
     break;
  case 's':
  case 'S':
     result = num1 - num2;
     break;
}
```

- *See Examples 5.13 and 5.14*