Chapter 4: Conditionals and Loops

CS 121

Department of Computer Science
College of Engineering
Boise State University

November 2, 2015

Chapter 4 Topics

- Flow of control

  Go to part 0

- Boolean expressions

  Go to part 1

- `if`, `else` and *block* statements

  Go to part 2

- Comparing data

  Go to part 3

- `switch` statements

  Go to part 4

- `while`, `do`, and `for` loops

  Go to part 5

- Iterators, ArrayLists

  Go to part 6

Flow of Control

- Statement execution is *linear* unless specified otherwise.
- To make our programs more interesting there are program statements that allow us to:
  - decide whether or not to execute a particular statement (conditional statements)
  - execute a statement over and over, repetitively (loops)
- These decisions are based on boolean expressions (or *conditions*) that evaluate to `true` or `false`
- The order of statement execution is called the flow of control

Conditional Statements

- A conditional statement lets us choose which statement will be executed next.
- Therefore they are sometimes called selection statements.
- Conditional statements give us the power to make basic decisions.
- Conditional statements in Java:
  - `if` statement
  - `if-else` statement
  - `switch` statement

The `if` statement

- The syntax of a basic `if` statement is:

```
if (condition)
    statement;
```

- The condition must be a boolean expression. It must return `true` or `false`. Note that the condition must be encolsed in parentheses.
- If the condition is `true`, then the statement is executed.
- If the condition is `false`, then the statement is skipped.

Equality and Relational Operators

- Often, conditions are based on equality operators or relational operators.

Operator	Meaning
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Note that the equality operator == is different than the assignment operator =

Conditions

Examples of `if` statements using equality and relational operators.

```
if (total == sum)
{
    System.out.println("total equals sum");
}
```

```
if (count > 50)
{
    System.out.println("count is more than 50");
}
```

```
if (letter != 'x')
{
    System.out.println("letter is not x");
}
```

```
if (s.charAt(0) ==  'A')
{
    System.out.println("String s starts with an A");
}
```

In-Class Exercise

Write an `if` statement that checks if the length of a `String` variable `str` is greater than zero.

Logical Operators

- Conditions can also use logical operators.

Op	Meaning	Example	Result
`!`	logical NOT	`!a`	`true` if `a` is `false`, `false` if `a` is `true`
`&&`	logical AND	`a && b`	`true` if `a` and `b` are both `true`, `false` otherwise
`||`	logical OR	`a || b`	`true` if `a` or `b` or both are `true`, `false` otherwise

- They all take boolean operands and produce boolean results.
- Logical NOT is a unary operator.
- Logical AND and logical OR are binary operators.

Logical Operators - Truth Tables

A Truth Table represents the values of a Boolean expression for all possible values of its inputs.

- Logical NOT

a	!a
false	true
true	false

- Logical AND and logical OR

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

Logical Operators and Expressions

- Expressions that use logical operators can form complex conditions.

```
if (total < MAX+5 && !found)
{
    System.out.println("processing...");
}
```

- All logical operators have lower precedence than the relational operators.
- Logical NOT has higher precedence than logical AND and logical OR.

Logical Operators and Expressions

- Specific expressions can be evaluated using truth tables.

```
if (total < MAX+5 && !found)
{
    System.out.println("processing...");
}
```

- Truth table:

total < MAX+5	found	!found	total < MAX+5 && !found
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Short-Circuited Operators

- The processing of logical AND and logical OR is short-circuited.
- If the left operand is sufficient to determine the result, the right operand is not evaluated.

```
if (count != 0 && total/count > MAX)
{
    System.out.println("Testing");
}
```

- If count is equal to 0, then we won't check the rest of the condition.
- This type of processing should be used carefully.

The `if` statement

- Consider the following if statement:

```
if (sum > MAX)
  delta = sum - MAX;
System.out.println("The sum is " + sum);
```

- First the condition is evaluated – the value of `sum` is either greater than the value of `MAX`, or it is not.
- If the condition is true, the assignment statement is executed – if it isn't, it is skipped.
- Either way, the call to `println` is executed at the end.
- Example: Age.java

Indentation

- The statement controlled by the if statement is indented to indicate that relationship.
- The use of a consistent indentation style makes a program easier to read and understand.
- Although it makes no difference to the compiler, **proper indentation is crucial for readability and maintainablity**.
- Remember, indentation is for the human reader, and is ignored by the computer. E.g., this is BAD:

```
if (total > MAX)
    System.out.println("Error!!");
    errorCount++;
```

- Despite what is *implied* by the indentation, the increment will occur whether the condition is true or not.

Block Statements

- Several statements can be grouped together into a block statement delimited by curly braces.
- A block statement can be used wherever a statement is called for in the Java syntax rules.

```
if (total > MAX)
{
    System.out.println("Error!!");
    errorCount++;
}
```

- To avoid confusion, it is best to *always* use block statements.

The `if-else` statement

- An `else` clause can be added to an `if` statement to make an `if-else` statement.

```
if (condition)
    statement1;
else
    statement2;
```

- If the condition is `true`, statement1 is executed.
- If the condition is `false`, statement2 is executed.
- One or the other will be executed, but never both.
- Examples: AgePhrases.java, Wages.java, Guessing.java

The Conditional Operator (1)

- Java has a conditional operator that uses a boolean condition to determine which of two expressions is evaluated.
- The syntax is

```
condition ? expression1 : expression2;
```

- If the condition is `true`, expression1 is evaluated.
- If the condition is `false`, expression2 is evaluated.
- The resulting value of the entire conditional operator is the value of the selected expression.

The Conditional Operator (2)

- The conditional operator is similar to an `if-else` statement, except that it is an expression that returns a value.
- For example:

```
int larger = ((num1 > num2) ? num1 : num2);
```

  - If `num1` is greater than `num2`, then `num1` is assigned to `larger`.
  - If `num1` is less than or equal to `num2`, then `num2` is assigned to `larger`.

- Here is another example:

```
System.out.println("Your change is " + count +
                   ((count == 1) ? "dime" : "dimes"));
```

Nested `if` Statements

- The statement executed as a result of an `if` statement or `else` clause could be another `if` statement.
- These are called nested `if` statements.
- An `else` clause is matched to the last unmatched `if` (no matter what the indentation implies).
- Braces should be used to specify the `if` statement to which an `else` clause belongs.
- Examples: MinOfThree.java
- **In-class exercise**: Write a code snippet to find the minimum of four numbers.

Comparing Data

- When comparing data using boolean expressions, it's important to understand the nuances of certain data types.
- Let's examine some key situations:
  - comparing floating point values for equality.
  - comparing characters.
  - comparing strings (alphabetical order).
  - comparing objects vs. comparing object references.

Comparing Floating Point Values (1)

- You should rarely use the equality operator (==) when comparing two floating point values (`float` or `double`).
- Two floating point values are equal only if their underlying binary representations match exactly.
- Computations often result in slight differences that may be irrelevant (e.g. 3.14 vs. 3.141592).
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal.

Comparing Floating Point Values (2)

- To determine the equality of two floating point values, we can use the following technique:

```
if (Math.abs(f1 - f2) < TOLERANCE)
{
    System.out.println("Essentially equal");
}
```

- If the difference between the two floating point values is less than the tolerance, they are considered to be equal.
- The tolerance could be set to any appropriate level. For example `10E-7` for `float` and `10E-15` for `double`.
- Example: TestDoubleCompare.java

Comparing Characters (1)

- Java character data is based on the Unicode character set. Unicode establishes a particular numeric value for each character, and therefore an ordering.
- We can use relational operators on character data based on this ordering.
- For example, the character `'+'` is less than the character `'J'` because it comes before it in the Unicode character set.
- Appendix C provides an overview of Unicode.

Comparing Characters (2)

- In Unicode, the digit characters (0-9) are contiguous and in order.
- Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in order.

Characters	Unicode Values
0-9	48 through 57
A-Z	65 through 90
a-z	97 through 122

- We can also add and subtract characters. For example:

```
System.out.println('b' - 'a');
System.out.println('9' - '0');
System.out.println('A' - 'a');
```

Comparing Strings (1)

- Recall that in Java a character string is an object.
- The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order.
- The `equals` method returns a boolean result.

```
if (name1.equals(name2))
{
    System.out.println("Same name");
}
```

Comparing Strings (2)

- We **cannot** use the relational operators to compare Strings.
- The `String` class contains a method called `compareTo` to determine if one string comes before another.
- Using the method would look something like:

```
name1.compareTo(name2)
```

  - returns zero if `name1` and `name2` are equal (contain the same characters).
  - returns a negative value if `name1` is less than `name2`.
  - returns a positive value if `name1` is greater than `name2`.

Comparing Strings (3)

```
if (name1.compareTo(name2) < 0)
{
    System.out.println(name1 + "comes first");
}
else if (name1.compareTo(name2) == 0)
{
    System.out.println("Same name");
}
else
{
    System.out.println(name2 + "comes first");
}
```

Lexicographic Ordering

- Because comparing characters and strings is based on a character set, it is called a lexicographic ordering.
- Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed.
- For example, the string `"Great"` comes before the string `"fantastic"` because all of the uppercase letters come before all of the lowercase letters in Unicode.
- Also, short strings come before longer strings with the same prefix (lexicographically). Therefore `"book"` comes before `"bookcase"`.

Comparing Objects vs. Comparing Object References

- The == operator can be applied to objects, but it returns `true` if the two references are *aliases* of each other. It doesn't compare the values of the objects.
- The `equals` method is defined for all objects, unless we redefine it when we write a class.
- By default, it will be the same as the == operator.
- It has been redefined in the `String` class to compare the characters in two strings.
- When writing classes, we can/should redefine the `equals` method to return true under the appropriate conditions.
- Example: StringEquals.java
- Example: PoetryPlay.java

In-class exercise

```
1  if (age < 18)
2  {
3      if(status == "happy")
4          System.out.println("Hi, I'm a minor and I'm happy!");
5      else if (status == "sad")
6          System.out.println("Hi, I'm a minor and I'm sad :(");
7      else
8          System.out.println("Hi, I'm a minor and I don't know my status");
9  }
10 else if (age >= 18 && age < 21)
11 {
12     System.out.println("Hey, I'm over 18, but still not 21.");
13 }
14 else
15 {
16     if(status == "happy")
17         System.out.println("I love getting older!");
18     else if(status == "sad")
19         System.out.println("Man, I'm getting old...");
20 }
21 System.out.println("Goodbye!");
```

- What is the output if age = 17 and status = "happy"?
- What is the output if age = 25 and status = "excited"?
- What is the output if age = 21 and status = "sad"?

The switch Statement (1)

- The switch statement provides another way to decide which statement to execute next.
- The general syntax of the switch statement is

```
switch (expression)
{
  case value1:
    statement-list1
  case value2:
    statement-list2
  case value3:
    statement-list3
  case ...
}
```

The `switch` Statement (1)

- A `switch` statement evaluates an expression, then attempts to match the result to one of several possible cases.
- Each `case` contains a value and a list of statements.
- The flow of control transfers to the statement associated with the first case value that matches.

The `switch` Statement (2)

- Often a `break` statement is used as the last statement in each case's statement list.
- A `break` statement causes control to transfer to the end of the `switch` statement.
- If a `break` statement is not used, the flow of control will continue into the next case. Sometimes this may be appropriate, but often we want to execute only the statements associated with one case.

The switch Statement (3)

- An example switch statement:

```
char option = 'A';

switch (option)
{
  case 'A':
    aCount++;
    break;
  case 'B':
    bCount++;
    break;
  case 'C':
    cCount++;
    break;
}
```

- **In-class Exercise**. Rewrite the above switch statement using if-else statements.

The `switch` Statement (4)

- A `switch` statement can have an optional `default case`.
- The `default` case has no associated value and simply uses the reserved word `default`.
- If the default case is present, control will transfer to it if no other case value matches.
- If there is no default case, and no other value matches, control falls through to the statement after the switch.

The switch Statement (5)

- Another example switch statement:

```
// Read a color from the user
String color = keyboard.nextLine().trim();

switch (color.toLowerCase())
{
    case "blue":
        countBlue++;
        break;
    case "green":
        countGreen++;
        break;
    case "purple":
        countPurple++;
        break;
    case "orange":
        countOrange++;
        break;
    default:
        System.out.println("Not in my top four!");
        break;
}
```

The `switch` Statement (6)

- The expression of a switch statement must result in an integral type, meaning an integer (`byte`, `short`, `int`, `long`), `char` or an `enum`.
- Switch statements can also use `String` type from Java 7 onward.
- It cannot be a boolean value or a floating point value (`float` or `double`).
- The implicit boolean condition in a switch statement is equality.
- You cannot perform relational checks with a switch statement.
- Example: GradeReport.java, FavoriteColors.java

`while`, `do`, and `for` loops

- Loops allow us to execute a statement multiple times.
- Like conditional statements, they are controlled by boolean expressions.
- Java has three kinds of loops:
  - the `while` loop
  - the `do` loop
  - the `for` loop

The `while` loop (1)

- A `while` loop has the following syntax:

```
while (condition)
{
  statement;
}
```

- If the condition is true, the statement is executed.
- The statement is executed repeatedly until the condition becomes false.
- If the condition of a while loop is false initially, the statement is never executed.
- Therefore, the body of a while loop will execute zero or more times.

The `while` loop (2)

- *While count is less than or equal to 5, print the value of count and increment the value of count by one.*

```
int count = 1;
while (count <= 5)
{
    System.out.println(count);
    count++;
}
```

- What is the output from the above code snippet?

The while loop (3)

- Write a loop that counts the number of times the letter 'z' occurs in a given String `s`.

```
String s = "I am a zizzer zazzer zuzz";
```

```
int count = 0;
int index = 0;
while (index < s.length())
{
    if (s.charAt(index) == 'z')
        count++;
    index++;
}
System.out.println("#z: " + count);
```

In-class exercise

- Write a `while` loop that prints the letters in a string variable `s`, one per line.

```
String s = "watch me go!";
```

The `while` loop (4)

- Let's look at some more examples of loop processing.
- A loop can be used to maintain a running sum. Or compute the average or min or max value from a series of values.
- A sentinel value is a special input value that represents the end of input.
  - Example: Average.java
- A loop can be used to validate input from a user.
  - Example: WinPercentage.java

Infinite Loops

- Infinite loops are loops that keep running forever. Usually, they are not good!
- Example of an infinite loop.

```
int count = 1;
while( count <= 25 )
{
    System.out.println (count);
    count--;
}
```

- To stop an infinite loop, interrupt your program execution with the cancel command (`ctrl-c`). In Eclipse, click on the red stop button.
- Infinite loops can be useful in certain circumstances.

```
while (true)
{
    //wait for interaction from user
}
```

- For example, the operating system runs in an infinite loop on your desktop, laptop or phone (unless you power it off or it crashes!)
- Example: InfiniteLoop.java

Nested Loops (1)

- Similar to nested if statements, loops can be nested as well. That is, the body of a loop can contain another loop. For each iteration of the outer loop, the inner loop iterates completely.
- How many times will the output be printed?

```
1  int count1 = 1;
2  while (count1 <= 10)
3  {
4      int count2 = 1;
5      while (count2 <= 50)
6      {
7          System.out.println ("Here again");
8          count2++;
9      }
10     count1++;
11 }
```

- What if the condition on outer loop was `(count1 < 10)`?
- What if the variable `count2` was initialized to 10 instead of 1 before the inner loop?

Nested Loops (2)

- A Palindrome is a string of characters that reads the same both forward and backward. Are the following palindromes?
  - radar
  - kayak
  - Radar
  - A man, a plan, a canal, Panama.
- Example: PalindromeTester.java
- Generalize to skip spaces, punctuation and changes in case for letters.

The do Loop (1)

- The do loop has the following syntax:

```
do
{
   statements;
}
while (condition);
```

- The statement is executed once initially, and then the condition is evaluated.
- The statement is executed repeatedly until the condition becomes false.
- The body of a do loop is executed at least once.

The do Loop (2)

- An example of a do loop:

```
int count = 0;
do
{
    count++;
    System.out.println (count);
}
while (count < 5);
```

- Example: ReverseNumber.java

`while` vs. `do-while`

- Check for understanding...what is the difference?

The `for` Loop (1)

- The `for` loop has the following syntax:

```
for (initialization; condition; increment)
    statement;
```

- The initialization is executed *once* before the loop begins.
- If the condition is true, the statement is executed, then the increment is executed.
- The condition is evaluated again, and if it is still true, the statement and increment are executed again.
- The statement and increment are executed repeatedly until the condition becomes false.

The for Loop (2)

- The for loop is functionally equivalent to the following while loop structure:

```
initialization;
while (condition)
{
    statement;
    increment;
}
```

The for Loop (3)

- An example of a for loop:

```
for (int count = 1; count <= 5; count++)
    System.out.println (count);
```

- The initialization section can be used to declare a variable.
- Like a while loop, the condition of a for loop is tested prior to executing the loop body.
- Therefore, the body of a for loop will execute zero or more times.
- A for loop is well suited for executing statements a specific number of times that can be calculated or determined in advance.

The for Loop (4)

- The increment section can perform any calculation.

```
for (int num = 100; num > 0; num = num - 5)
    System.out.println (num);
```

- Write a for loop to print the multiples of 3 from 3 to 300.

```
for (int i = 1; i <= 100; i++)
{
    System.out.println(3*i);
}
```

- Write a for loop to print the multiples of 3 from 300 down to 3.

```
for (int i = 100; i >= 1; i--)
{
    System.out.println(3*i);
}
```

The for Loop (5)

- Write a for loop that computes the sum of integers from 20 to 70, inclusive, and then prints the result.

```
int sum = 0; int low = 20; int high = 70;
for (int i = low; i <= high; i++)
{
    sum += i;
}
System.out.println("sum = " + sum);
```

- Write a for loop that creates a new string composed of every other character from the String object called name

```
String s = "";
for (int i = 0; i < name.length(); i += 2)
{
    s += name.charAt(i);
}
System.out.println(s);
```

The for Loop (6)

- Example: Multiples.java.
  - Shows how to print fixed number of values per line using the mod % operator inside a for loop.
- Nested for loops are similar to other nested loops. What does the following loop print?

```
//PP 4.6
int n = 12;
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= n; j++)
    {
        System.out.print(i*j + " ");
    }
    System.out.println();
}
```

In-class Exercise

- What does the following `for` loop print?

```
final int MAX_ROWS = 10;

for (int row = MAX_ROWS; row > 0 ; row--)
{
    for (int star = 0; star < row; star++)
        System.out.print ("*");
    System.out.println();
}
```

The `for` Loop (7)

- Each expression in the header of a `for` loop is optional.
- If the initialization is left out, no initialization is performed.
- If the condition is left out, it is always considered to be true, and therefore creates an infinite loop.
- If the increment is left out, no increment operation is performed.
- The following is a valid, infinite `for` loop!

```
for (;;) {}
```

More `for` Loop Examples

- More examples using `for` loops.
  - Example: RandomBoxes.java
  - Example: BullsEye.java
  - Example: BullsEyeScalable.java

In-class Exercise

- What does the `for` loop in following `paintComponent` method draw?

```
private final int SIZE = 30;
private final int GAP = 10;
private final int WIDTH = 800;

public void paintComponent(Graphics page)
{
  super.paintComponent(page);
  int n = WIDTH / 40;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      page.setColor(Color.blue);
      page.fillRect((SIZE + GAP)*i,
                    (SIZE + GAP)*j,
                     SIZE, SIZE);
    }
}
```

- A two-dimensional lattice of blue squares!

Iterators (1)

- An iterator is an object that allows you to process a collection of items one at a time.
- It lets you step through each item in turn and process it as needed.
- An iterator object has a `hasNext` method that returns true if there is at least one more item to process.
- The `next` method returns the next item.
- Iterator objects are defined using the `Iterator` interface.

Iterators (2)

- Some classes in the Java API are iterators. For example, the `Scanner` class is an iterator.
  - the `hasNext` method returns true if there is more data to be scanned.
  - the `next` method returns the next scanned token as a string.
- The `Scanner` class also has variations on the `hasNext` method for specific data types (such as `hasNextInt`).

The ArrayList Class

- The ArrayList class stores a list of objects. It is part of the java.util package.
- It grows and shrinks as needed. Each object in it has a numeric index, starting from zero. Objects can be inserted or removed and the indices adjust accordingly.
- The declaration establishes the type of objects that a given ArrayList class can store. This is an example of generics.

```
ArrayList<String> band = new ArrayList<String>();
```

```
band.add("Paul");
band.add("Pete");
band.add("John");
band.add("George");

System.out.println("Size of the band: " + band.size());
```

- The ArrayList class cannot store primitive types. We can use wrapper objects if we want to store primitive types in an ArrayList.

Iterators and the for-each Loop

- A variant of the `for` loop simplifies the repetitive processing for any object that implements the `Iterable` interface.
- This style of `for` loop is referred to as the `for-each` loop.
- An `Iterable` interface provides an `iterator`.
- For example, an `ArrayList` is an `Iterable` list that we can use with a for-each loop.

```
//Iterate over the band members using a for-each loop
for (String member: band) {
    System.out.print(member + " ");
}
```

- It can be read: "for each `member` in the list of band members"
- Example: Beatles.java
- The for-each loop will also be helpful when processing arrays (Chapter 7).

In-class Exercise

- Write a code snippet that creates a 100 random colors and adds them to an `ArrayList`.
- Then write a *for-each* loop that walks through the colors and finds the one with the maximum red component.

Review

- `while`, `do-while`, and `for` loops
- Which loop should we use to
  - Print numbers 1 - 100.
  - Keep asking the user to enter input until they enter a specific sentinel value.
  - Ask the user for 3 items.
  - Read a number from the user and store each digit of the number in a separate int.
  - Reverse a String.

Exercises

- Read Chapter 4.
- **Recommended Homework**:
  - Exercises: EX 4.2–4.4, 4.6, 4.8, 4.11–4.14, 4.17, 4.21, 4.22.
  - Projects: PP 4.3, 4.10, 4.12.
- Browse: Sections 5.1–5.4.