Java By Abstraction: Chapter 5

Control Structures

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

Flow of Control

- Previous chapters illustrated sequential flow
- Altering execution flow can result in powerful data processing
- Selective flow control:
  - Execution path takes one of many branches
- Iterative flow control:
  - Execution path repeats until a condition is met

Review: Boolean Operators

- Relational: < > <= >= == !=
  - 0 < x < 1 // incorrect, syntax error
  - x > 0 && x < 1 // valid syntax

- Logical NOT: !
- Logical AND: &&
- Logical OR: ||

Lazy (Short-Circuit) Evaluation

- Applies to && and || operators
- Does not evaluate second operand unless necessary

  false && *p* == false // regardless of value of *p*

  true || *p* == true // regardless of value of *p*

  Thus, *p* is never evaluated
- Example

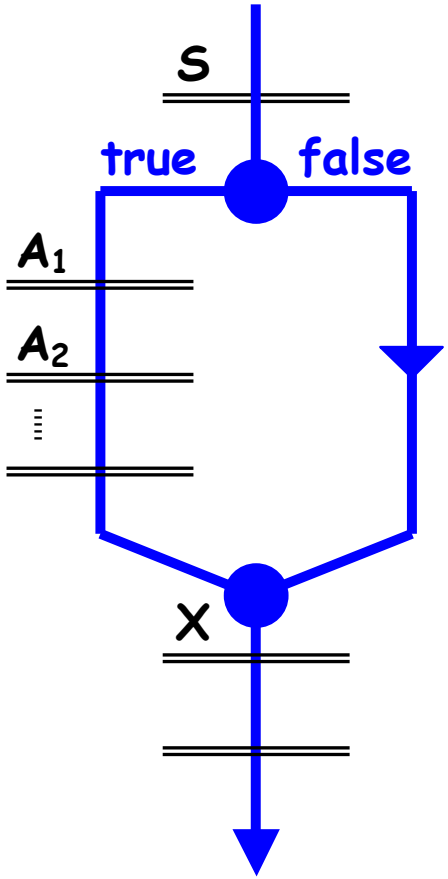
  x.equals(y) // results in exception if x is null

  x != null && x.equals(y) // evaluated iff x is not null

Selection (a.k.a. Branching)

- Involves the evaluation of a Boolean expression
- If the expression evaluates to true, code execution takes a separate path
- In Java, the separate path is enclosed in a code block (indicated by braces)
- If the expression evaluates to false, code execution continues with the statement after the code block

if Statement




```
Statement-S
if (condition)
{
    Statement-A1
    Statement-A2
        ⋮
}
Statement-X
  ⋮
```

Pitfall: Including a Semicolon

- Example

```
int entry = input.nextInt();
int absValue = entry;
if (entry < 0);
{
   absValue = -entry;
}
output.println(absValue);
```

- Consequently, the entry will always be negated

Pitfall: Omission of Braces

- Example

```
if (count > maximum)
        count--;
        output.println("Maximum exceeded.");
```

- Count will be decremented if the condition is true
- Print statement will be executed regardless

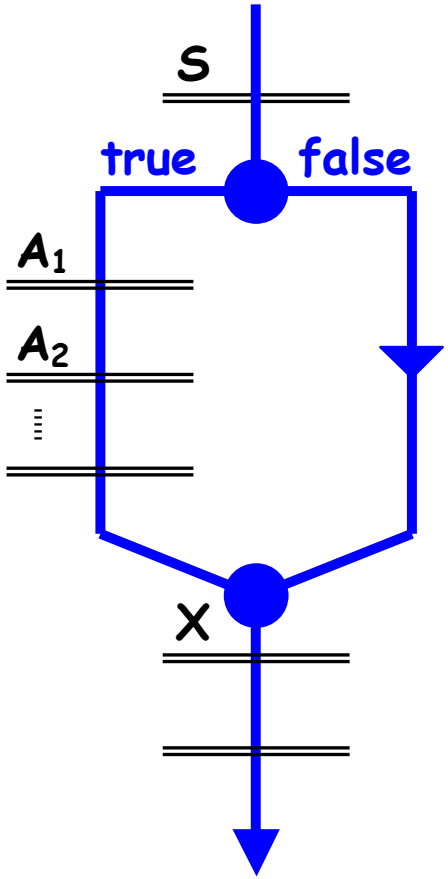
Pitfall: Variable out of Scope

- Variables declared in a code block are accessible only within that block
- Example

```
if (entry < 0)
{
   int absValue = -entry;
}
output.println(absValue);
```

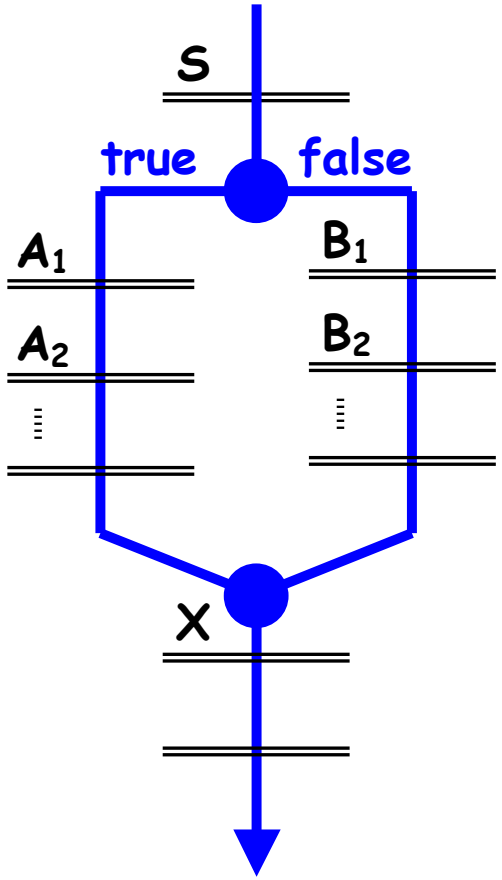
- Variable absValue is not accessible outside the block
- Results in a compile time error

if Statement (Recall)




```
Statement-S
if (condition)
{
    Statement-A1
    Statement-A2
        ⋮
}
Statement-X
  ⋮
```

if-else Statement



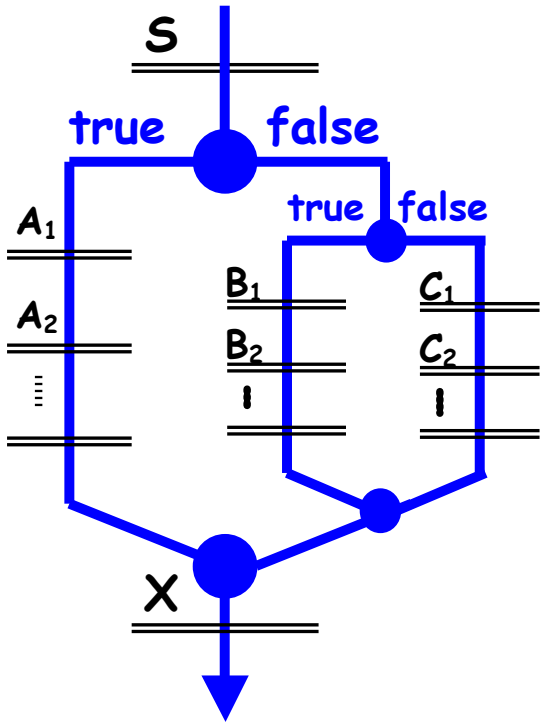

```
Statement-S
if (condition)
{
    Statement-A1
    Statement-A2
    ...
} else
{
    Statement-B1
    Statement-B2
    ...
}
Statement-X
```

if or if-else?

- Beneficial to use "if-else" statements
  - Clearly represents "decision making" choices
  - Aids in debugging logic errors

- Better to use "if" statements if "else" block is empty

Multiple Conditions (if, else-if, else)




```
Statement-S
if (condition-1)
{  Statement-A1
   Statement-A2
   ...
} else if (condition-2)
{  Statement-B1
   Statement-B2
   ...
} else
{  Statement-C1
   Statement-C2
   ...
}
Statement-X
```

switch Statement

```
switch (var)
{   case value1:
        ... // S1
        break;
    case value2 value3:
        ... // S2
        break;
    default:
        ... // S3
        break;
}
```

```
if (var == value1)
{   ... // S1
} else if (var == value2 ||
           var == value3)
{   ... // S2
} else
{   ... // S3
}
```

Prior to Java 1.7, *var* had to be an `int` variable.
As of Java 1.7, *var* can be an `int`, `char`, or String.

Iteration

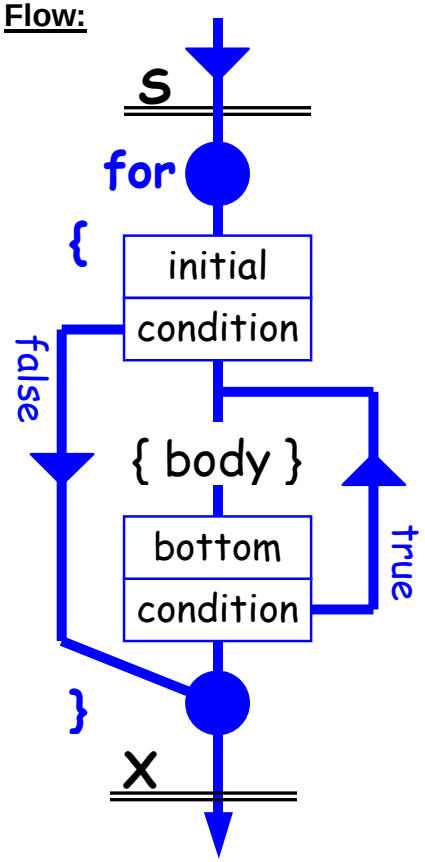
- Computer can execute millions of instructions in a second
- But, programmers don't need to specify each instruction individually
- Iteration allows a block of code to be executed repeatedly
- Accomplished using loop structure

for Loop

- Loop body
  - Statements to be executed iteratively (i.e., to be looped)
- Initialization statement (optional)
  - Executed once, when the loop is first encountered
  - Used to declare and/or initialize any variables use within the loop body (be careful of variable scope)
- Boolean condition to continue iteration (i.e., looping)
  - Similar to the if-statement condition
  - Loop body is executed if the condition holds (i.e., is true)
- Update statement (optional)
  - Update variables/state at the end of each iteration (i.e., loop)

for Loop

Flow:



Syntax:

```
Statement-S
for (initial; condition; bottom)
{
    body;
}
Statement-X
```

Algorithm:

1. Start the `for` scope
2. Execute `initial`
3. If `condition` is `false` go to 9
4. Start the body scope {
5. Execute the body
6. End the body scope }
7. Execute `bottom`
8. If `condition` is `true` go to 4
9. End the `for` scope

Example

- Output the numbers from 1..100
- Sequential

```
System.out.println("1");
System.out.println("2");
System.out.println("3");
…
```

- Iterative

```
final int MAX = 100;
for (int count = 1; count <= MAX; count++)
{
   System.out.println(count);
}
```

Importance of Loop Condition

- Can be as simple or complex as necessary
- If false before first iteration, loop skipped
- If always true, loop continues indefinitely

Sentinel-Based Input

- Sentinel: a value used to signal the end of input
- Task:
  - Read-in positive integers as input
  - Input -1 to signal end of input
  - Output sum of inputs

Sentinel-Based Input (code)

```
final int SENTINEL = -1;
int sum = 0;
output.println("Enter positive integers to add (-1 to quit):");
for (int num = input.nextInt(); num != SENTINEL;
                num = input.nextInt())
{
        if (num > 0)
        {
                sum += num;
        }
}
output.println("Sum: " + sum);
```

Friendly Input Validation

```
…
output.print("Enter a positive integer: ");
int n;
for (n = input.nextInt(); n <=0; n =
  input.nextInt())
{
  output.print("Invalid! Please retry: ");
}
…
```

Nested Loops

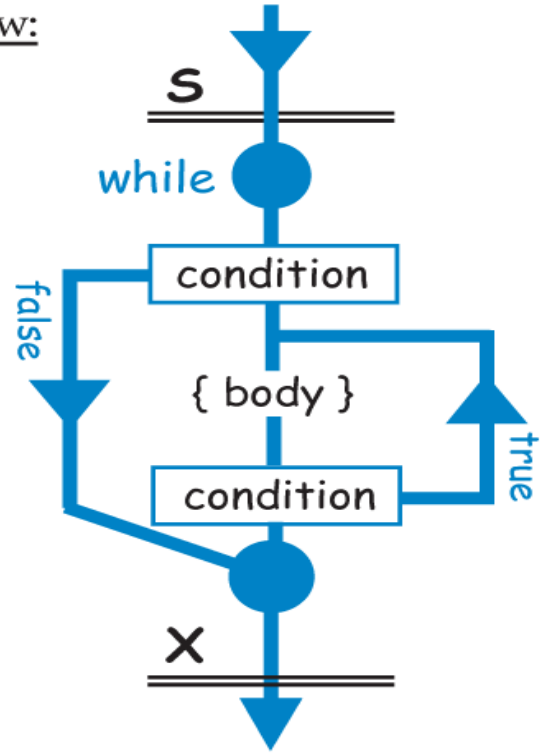
```
final int M = 5;
final int N = 3;
for (int i = 0; i < M; i++)
{
  for (int j = 0; j < N; j++)
  {
      output.println("" + i + " "
  + j);
  }
}
```

Output (p. 195):

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
...
```

while Loop

Flow:



Syntax:

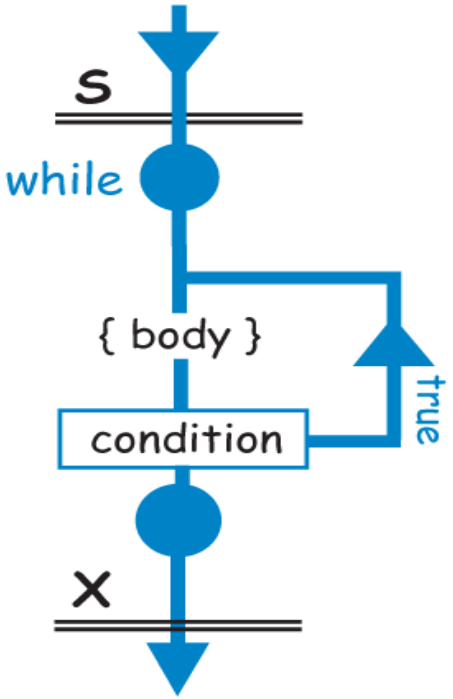
```
Statement-S
while(condition)
{
    body;
}
Statement-X
```

Algorithm:

1. If `condition` is `false` go to 6
2. Start the body scope {
3. Execute the body
4. End the body scope }
5. If `condition` is `true` go to 2
6. End the `while` scope

do-while Loop

Flow:



Syntax:

```
Statement-S
do
{
    body;
}
while (condition)
Statement-X
```

Algorithm:

1. Start the body scope {
2. Execute the body
3. End the body scope }
4. If `condition` is `true` go to 1
5. End the `while` scope

Lots of Loops

- while ↔ for:

```
while
   (condition)
{
   …
}

for (; condition;
   )
{
   …
}
```

- do-while ↔ for:

```
do
{
   …
} while (condition);

for (boolean b = true; b; b =
   condition)
{
   …
}
```

File Input/Output

- I/O from user:
  - Scanner input = new Scanner(System.in);
  - PrintStream output = new PrintStream(System.out);
- I/O from a file:
  - Scanner fileInput = new Scanner(new File("log.txt");
  - PrintStream fileOutput = new PrintStream("log.txt");
- Change main method signature (very important):
  - …(String[] args) throws java.io.IOException

Exercises

1. Output the multiplication table for 1..10. Modify your code so that it reads in from a file and outputs to a file.
2. Write code to output the nth number in the Fibonacci sequence.
3. Output the factors of a positive integer.
4. Use a loop to determine if a positive integer, n, is prime. Do so by checking if it is evenly divisible by 2..n/2.
5. Write code to output an * that zig-zags across the screen (width 20 characters).