

Chapter 5:

Program Logic &

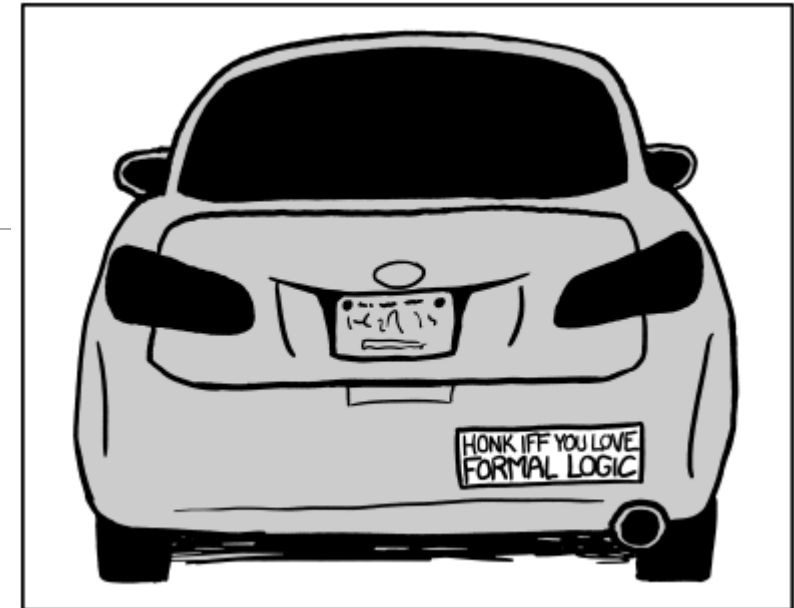
Indefinite Loops

WEEK 7:

THE WHILE LOOP

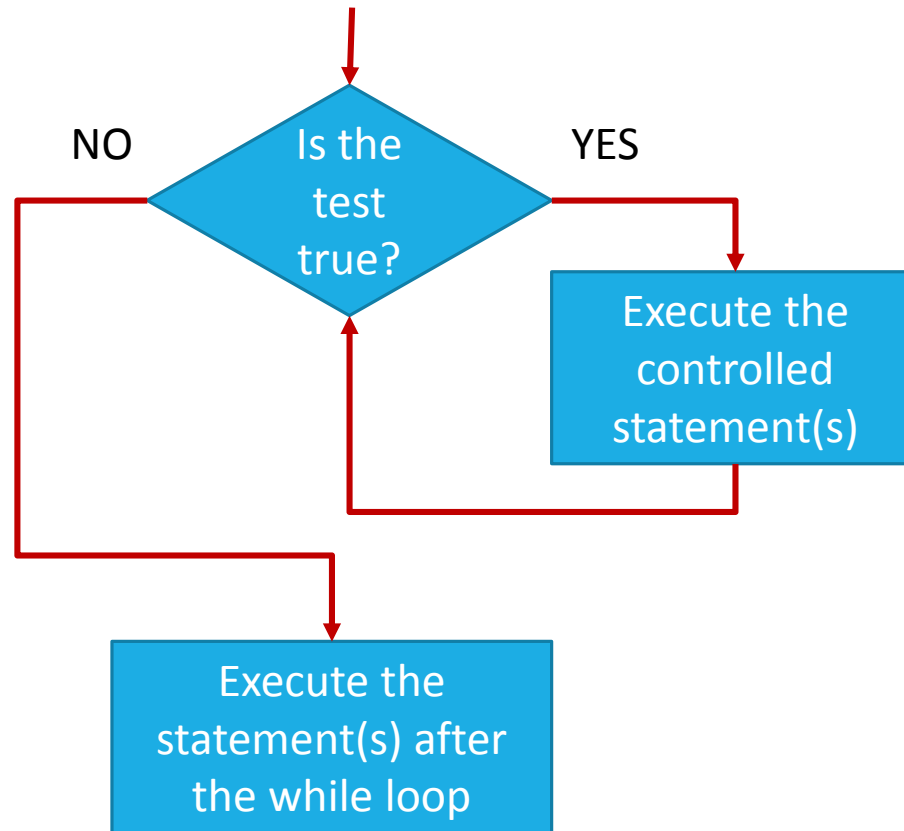
THE BOOLEAN TYPE

USER ERRORS



The while Loop

- You were introduced to the for loop in Chapter 2. The for loop is an extremely useful and commonly used control structure. Sometimes, however, you won't know how many loop iterations you need.
- The while loop is an example of an INDEFINITE LOOP – it will keep executing as long as its test condition keeps evaluating to “true”.
- **Flow of a while loop:**



```
while(<test>){  
    <statement>;  
    <statement>;  
    . . .  
    <statement>;  
}
```

// Simple Example

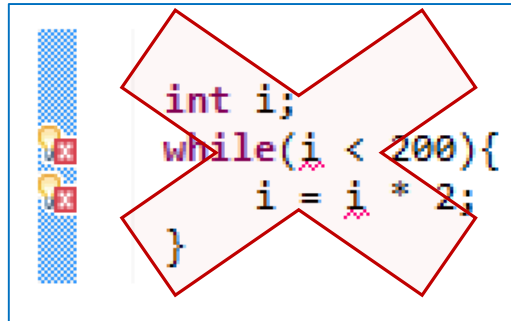
```
int i = 1;  
while(i < 200){  
    i = i * 2;  
}
```

Priming the while Loop

- Looking at our simple example, you'll note that the variable "i" is declared and initialized *just before* the while loop:

```
// Simple Example
```

```
int i = 1;  
while(i < 200){  
    i = i * 2;  
}
```



Forgetting to initialize the variable results in an error:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:  
The local variable i may not have been initialized  
The local variable i may not have been initialized
```

- Priming a Loop:** Initializing variable **BEFORE** a loop to ensure that the loop is entered.
- Using the initial value of **one**, what will the end result of this **while** loop be? And how many iterations will take place?
- Essentially the value of i keeps doubling with each loop iteration, until the test condition is not met.
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 256$
- After the 8th loop iteration, the test condition evaluates to FALSE, and the loop is exited.

Infinite Loops



- It is easy to inadvertently create infinite loops when writing while loops
- For example, consider the following loops – both end up looping forever:

```
int i = 1;
while(i < 200){
    System.out.println("Hello!");
}
```

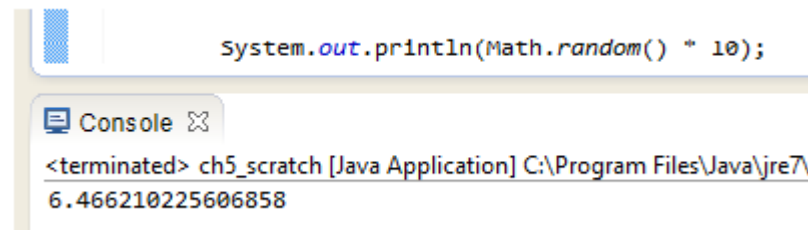
This loop has no “**update condition**,” so the value of “i” remain 1 (which is less than 200) forever.

```
int j = 100;
while(j > 1){
    System.out.println("Hello!");
    j = j + 1;
}
```

This loop has an “**update condition**,” but it makes the value of “j” *larger*! (The programmer probably intended to use a **minus** sign, not a plus sign).

Random Numbers

- Java provides a number of ways of obtaining **pseudorandom numbers**
- **Pseudorandom Numbers:** Numbers derived from predictable and well-defined algorithms that MIMIC the properties of numbers chosen at random
- One of the ways to obtain a pseudorandom number is to use the **random** method from the *Math* class
- This method returns a value of type double in the range: $0.0 \leq \text{Math.random()} < 1.0$
- By using multiplication, the range of numbers provided by this function can be changed
 - **For example**, multiplying by 10 will give a number in the range $0.0 \leq \text{Math.random()} * 10 < 10.0$

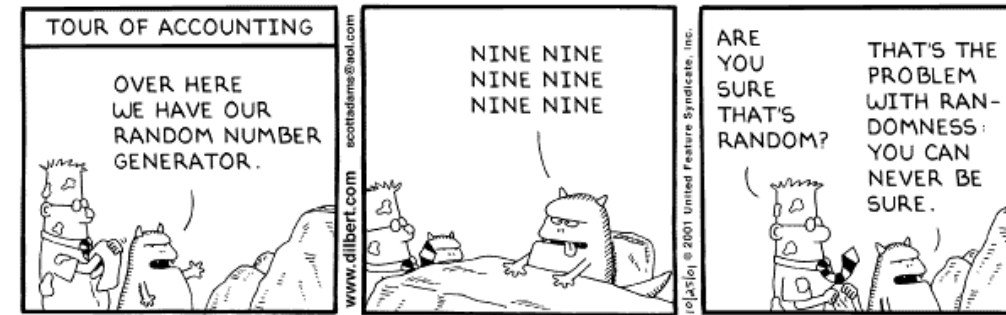


```
System.out.println(Math.random() * 10);
```

Console

<terminated> ch5_scratch [Java Application] C:\Program Files\Java\jre7\6.466210225606858

More Random Numbers



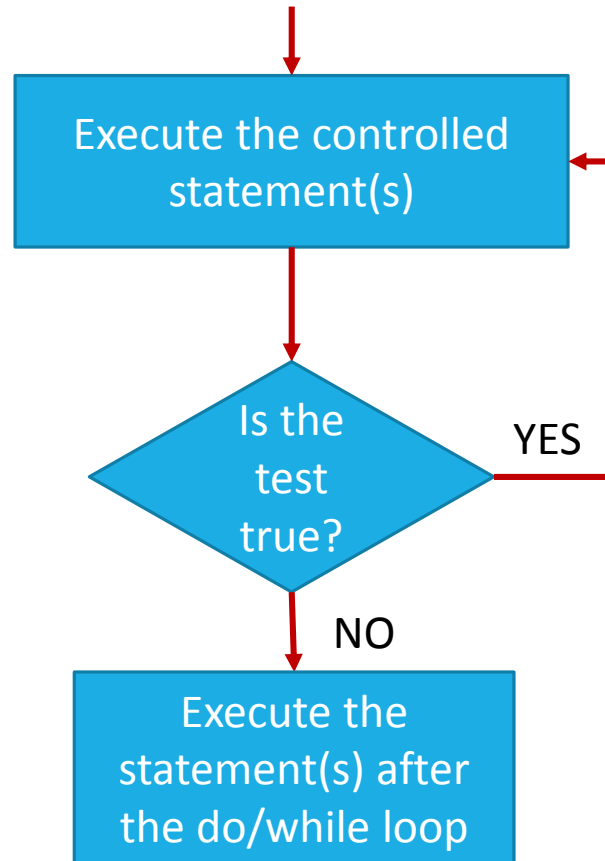
Copyright © 2001 United Feature Syndicate, Inc.

- Another way to obtain random numbers in Java is by using Java's *Random* class
- To use this class, you must include an import statement such as `"import java.util.*;"`
- You can use this class to create *Random* objects in your programs: `Random r = new Random();`
- The table below lists some useful methods of Random objects:

Method	Description
<code>nextInt()</code>	Random integer between -2^{31} and $(2^{31} - 1)$
<code>nextInt(max)</code>	Random integer between 0 and $(\text{max} - 1)$
<code>nextDouble()</code>	Random real number between 0.0 (inclusive) and 1.0 (exclusive)
<code>nextBoolean()</code>	Random logical value of true or false

The do/while Loop

- An alternative to a while loop that checks the loop condition AFTER the first iteration of the loop (versus the while loop, which checks the loop condition BEFORE entering the loop)
- Since the do/while loop checks the test condition at the “end” of the loop, it will always execute AT LEAST ONCE
- Flow of control:



```
do{  
    <statement>;  
    <statement>;  
    . . .  
    <statement>;  
}while(<test>);
```

```
// Simple Example  
  
double i;  
do{  
    double a = Math.random() * 10;  
    double b = Math.random() * 10;  
    i = a + b;  
    System.out.println("Hello! i = " + i);  
}while(i > 5 && i < 15);
```

Fencepost Algorithms



- A common programming problem is printing out entries in a list with some character (often a comma) separating them.
- How do we do this without having an extra “connecting character” at the end of the list?
- **Example:**

```
Random r = new Random();

int[] numList = new int[10];
for(int i = 0; i < 10; i++){
    numList[i] = r.nextInt(10);
}
for(int j = 0; j < 10; j++){
    System.out.print(numList[j] + ",");
}
```

```
<terminated> ch5_scratch [Java]
1,7,3,4,3,8,7,5,6,7,
```

Solution – print the first character outside the loop:

```
Random r = new Random();

int[] numList = new int[10];
for(int i = 0; i < 10; i++){
    numList[i] = r.nextInt(10);
}
System.out.print(numList[0]);
for(int j = 1; j < 10; j++){
    System.out.print(", " + numList[j]);
}
```

```
<terminated> ch5_scratch [Java]
7,3,8,6,8,4,8,3,3,0
```


Sentinel Loops

- **Sentinel:** A special value that signals the end of input.
- You might want to write a program that reads user input until a specific value is entered.
- For example, what if you wanted to create a program that accepted numerical input and then found the sum of the numbers entered? You could use “-1” as a sentinel
- This only works if you don’t want to allow the user to enter negative numbers...

In pseudocode:

```
sum = 0
Prompt user and read number
while (we haven't seen the sentinel value){
    add the number to sum
    prompt user and add number
}
```

In Java code:

```
Scanner console = new Scanner(System.in);
int sum = 0;
System.out.println("next integer (-1 to quit): ");
int number = console.nextInt();
while(number != -1){
    sum += number;
    System.out.println("next integer (-1 to quit): ");
    number = console.nextInt();
}
System.out.println("The sum is: " + sum);
```

The boolean Type

- Java type **boolean** is used to describe logical true/false relationships.
- The **boolean** type is a Java primitive.
- We have already been using Boolean expressions in our test conditions for control structures (loops and conditionals).
- Methods may return Boolean results – for example, the String class has the following methods that return a Boolean result:
 - `startsWith()`
 - `endsWith()`
 - `equals()`
 - `equalsIgnoreCase()`

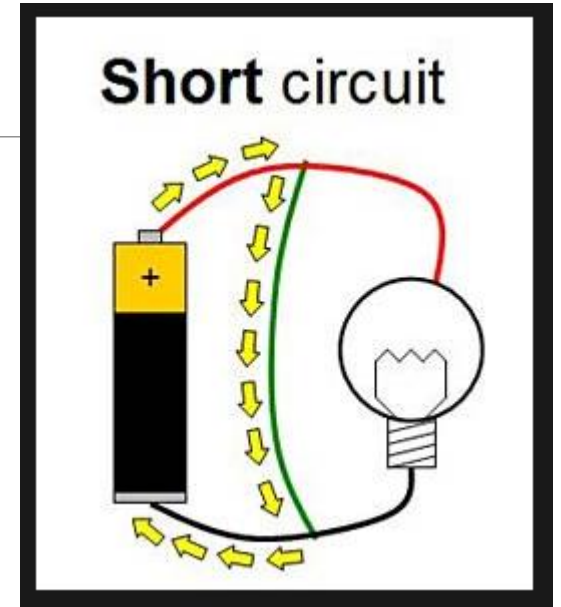
Logical Operators

- In Java, you can form complicated Boolean expressions using what are known as the logical operators (shown in the table)
- The NOT operator reverses the truth value of its operand
- The AND operator and the OR operator tie two Boolean expressions together, creating a new Boolean expression
- Logical operators are especially useful when the condition to be tested cannot be reduced to a single expression (e.g. when checking if a number falls within a certain range)

Operator	Meaning	Example	Value
&&	AND (conjunction)	(2 == 2) && (3 < 4)	true
	OR (disjunction)	(1 < 2) (2 == 3)	true
!	NOT (negation)	!(2 == 2)	false

Short-circuited Evaluation

- Short-circuited Evaluation: The property of the logical operators `&&` and `||` that prevents the 2nd operator from being evaluated if the overall result is obvious from the value of the first operand.
- For `&&`, if the first operand evaluates to false, the 2nd operand is never evaluated.
- For `||`, if the first operand is true, the 2nd operand is never evaluated.



NOTE ON OPERATOR PRECEDENCE: Logical `&&` and logical `||` have lower precedence than equality operators and relational operators. This means that the operands of logical expressions will be evaluated BEFORE the logical testing for conjunction/disjunction (`&&/||`) is done.

boolean Variables



- All if and if/else statements are controlled by Boolean tests
- These tests can be boolean variables or Boolean expressions
- If a Boolean test is assigned to a variable, it can be used more than once in your program
- For example, consider the following code fragment:


```
boolean temperatureNice = (degrees > 60 && degrees < 85);  
boolean rainfallNone = (inchesRain == 0);  
boolean socialDay = (friendsSeen > 2);  
boolean goodDay = (temperatureNice && rainfallNone && socialDay);
```

boolean Flags

- **boolean flag:** A special kind of boolean variable – often used within a loop to record error conditions or to signal completion
- For example, you might want a flag inside a loop that calculates a cumulative sum to alert you if the sum ever becomes negative:

```
double sum = 0.0;
boolean negative = false;
for(int i = 1; i <= 20; i++){
    System.out.print("Enter a number: ");
    double next = console.nextDouble();
    sum += next;
    if(sum < 0){
        negative = true;
    }
}
System.out.println("sum = " + sum);
if(negative){
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

Notice how you can just use the variable name as the test – the variable stores either “true” or “false” so this test is essentially checking if **negative** is storing the value “true”?



Negating Boolean Expressions

- The logical NOT operator can be used to negate Boolean expressions
- There are a few ways to accomplish this:
 1. The entire Boolean expression can be enclosed in parentheses and negated
 2. The expression can be “simplified” using De Morgan’s laws
- **Examples:**
 1. `while(!(num ≥ 1 && num ≤ 100))` – evaluates to true only if num is **NOT** in the range $1 \leq \text{num} \leq 100$
 2. `while(num < 1 || num > 101)` – performs the **same** test
- Simplifying the expression is the “preferred” method
- For this class, you will not be penalized if you use the first method – as long as your program performs the test you intend it to, you are good to go. Feel free to use whichever method seems most appropriate for your situation.

De Morgan's Laws

- Instead of negating an entire Boolean expression, the same effect may be accomplished by negating each operand individually and changing the logical operator in the expression

Original expression	Negated expression	Simplified negation
<code>p q</code>	<code>!(p q)</code>	<code>!p && !q</code>
<code>p && q</code>	<code>!(p && q)</code>	<code>!p !q</code>

Examples: Original Expressions	Negated/Simplified expressions (there are other correct forms for these – see if you can think of some...)
<code>boolean temperatureNice = (degrees > 60 && degrees < 85);</code>	<code>boolean temperatureNasty = (degrees <= 60 degrees >= 85);</code>
<code>boolean rainfallNone = (inchesRain == 0);</code>	<code>boolean rainfall = (inchesRain > 0);</code>
<code>boolean socialDay = (friendsSeen > 2);</code>	<code>boolean antisocialDay = (friendsSeen <= 2);</code>
<code>boolean goodDay = (temperatureNice && rainfallNone && socialDay);</code>	<code>boolean badDay = !goodDay;</code>

User Errors



- Earlier, you learned that it is good programming practice to think about preconditions to your methods
- Programs that use input obtained from users CANNOT assume that preconditions will be met
- Thus, programs should be written to deal with USER ERRORS – i.e. programs should be written to be ROBUST
- Robust: The ability of a program to execute even when presented with illegal data.
- Programs can be made robust by checking user input. Boolean tests can be very effective in such checks.

Handling User Errors

- A common programming task is asking the user for integer input.
- What if the value entered is NOT an integer? The code shown demonstrates how such a situation may be handled gracefully without causing your program to fail.

SAMPLE OUTPUT

```
Please enter an integer: hi
Not an integer -- try again.
Please enter an integer: 11.0
Not an integer -- try again.
Please enter an integer: 5

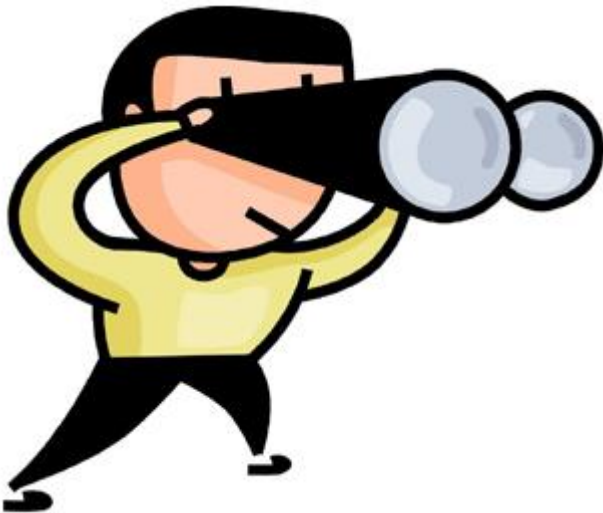
Thank you for the 5
```

```
public class userError {

    public static void main(String[] args) {
        Scanner console = new Scanner (System.in);
        String prompt = "Please enter an integer: ";
        int myInt = getInt(console, prompt);
        System.out.println();
        System.out.println("Thank you for the " + myInt);
    }

    private static int getInt(Scanner console, String prompt) {
        System.out.print(prompt);
        while(!console.hasNextInt()){
            console.next(); // to discard the bad input
            System.out.println("Not an integer -- try again.");
            System.out.print(prompt);
        }
        return console.nextInt();
    }
}
```

Scanner Lookahead



- The Scanner class has methods that let you perform a test BEFORE actually reading a value
- This means that you can ask the user for a certain type of input (e.g. an int) and you can check that the input received was, in fact, an int BEFORE attempting to use the input in your program
- These tests evaluate to “true” or “false” (i.e. they are Boolean tests)

Scanner class method – used with a Scanner object named console	Test performed
<code>console.hasNext();</code>	Checks if there is a next token
<code>console.hasNextInt();</code>	Checks if there is a next integer token
<code>console.hasNextDouble();</code>	Checks if there is a next double token

Concluding Comments

- **while loop:** A while loop can be used to write indefinite loops that keep executing until some condition fails.
- **priming a loop:** setting the values of variables that will be used in the loop test, so that the test will be sure to succeed the first time and the loop will execute.
- **pseudorandom numbers:** Numbers that mimic random numbers – can be generated using `Math.random()` or by using objects of the `Random` class.
- **do/while loop:** A variation on the while loop that checks the loop condition at the end of the loop body, meaning that the loop will always execute at least once.
- **boolean primitive:** represents the logical values of either “true” or “false”.
- **Boolean expressions:** use relational operators in combination with logical operators to check for specific conditions.
- **Logical && operator:** evaluates to true if both operands individually evaluate to true.
- **Logical || operator:** evaluates to true if either operand evaluates to true.
- **robust:** term used to describe a program that can handle illegal data (such as bad user input) without failing