Building Java Programs

Chapter 5
Lecture 5-4: More `boolean`, Assertions, `do/while` loops

reading: 5.3, 5.4, 5.1

Type `boolean`

reading: 5.3

Recap: Type `boolean`

- **`boolean`**: A logical type whose values are `true` and `false`.
  - A **test** in an `if`, `for`, or `while` is a `boolean` expression.

```
boolean minor = (age < 21);

if (minor) {
    System.out.println("Can't purchase alcohol!");
}
```

  - `boolean` expressions can be combined using *logical operators*:

Operator	Description	Example	Result
`&&`	and	`(2 == 3) && (-1 < 5)`	`false`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(2 == 3)`	`true`

"Short-circuit" evaluation

- Java stops evaluating a test if it knows the answer.
  - `&&` stops early if any part of the test is `false`
  - `||` stops early if any part of the test is `true`
- The following test will crash if s2's length is less than 2:

```
// Returns true if s1 and s2 end with the same two letters.
public static boolean rhyme(String s1, String s2) {
    return s1.endsWith(s2.substring(s2.length() - 2)) &&
           s1.length() >= 2 && s2.length() >= 2;
}
```

- The following test will not crash; it stops if length < 2:

```
// Returns true if s1 and s2 end with the same two letters.
public static boolean rhyme(String s1, String s2) {
    return s1.length() >= 2 && s2.length() >= 2 &&
           s1.endsWith(s2.substring(s2.length() - 2));
}
```


De Morgan's Law

- **De Morgan's Law**: Rules used to negate boolean tests.
  - Useful when you want the opposite of an existing test.

Original Expression	Negated Expression	Alternative
`a && b`	`!a || !b`	`!(a && b)`
`a || b`	`!a && !b`	`!(a || b)`

  - Example:

Original Code	Negated Code

`if (x == 7 && y > 3) {`
`  ...`
`}`
	`if (x != 7 || y <= 3) {`
	`  ...`
	`}`

Boolean practice questions

- Write a method named `isVowel` that returns whether a `String` is a vowel (a, e, i, o, or u), case-insensitively.
  - `isVowel("q")` returns `false`
  - `isVowel("A")` returns `true`
  - `isVowel("e")` returns `true`
- Change the above method into an `isNonVowel` that returns whether a `String` is any character except a vowel.
  - `isNonVowel("q")` returns `true`
  - `isNonVowel("A")` returns `false`
  - `isNonVowel("e")` returns `false`

Boolean practice answers

```
// Enlightened version.  I have seen the true way (and false way)
public static boolean isVowel(String s) {
    return s.equalsIgnoreCase("a") || s.equalsIgnoreCase("e") ||
           s.equalsIgnoreCase("i") || s.equalsIgnoreCase("o") ||
           s.equalsIgnoreCase("u");
}


// Enlightened "Boolean Zen" version
public static boolean isNonVowel(String s) {
    return !s.equalsIgnoreCase("a") && !s.equalsIgnoreCase("e") &&
           !s.equalsIgnoreCase("i") && !s.equalsIgnoreCase("o") &&
           !s.equalsIgnoreCase("u");

    // or, return !isVowel(s);
}
```


When to return?

- Methods with loops and return values can be tricky.
  - When and where should the method return its result?
- Write a method `seven` that accepts a `Random` parameter and uses it to draw up to ten lotto numbers from 1-30.
  - If any of the numbers is a lucky 7, the method should stop and return `true`. If none of the ten are 7 it should return `false`.
  - The method should print each number as it is drawn.

```
15 29 18 29 11 3 30 17 19 22      (first call)
29 5 29 4 7                       (second call)
```


Flawed solution

```
// Draws 10 lotto numbers; returns true if one is 7.
public static boolean seven(Random rand) {
    for (int i = 1; i <= 10; i++) {
        int num = rand.nextInt(30) + 1;
        System.out.print(num + " ");
        if (num == 7) {
            return true;
        } else {
            return false;
        }
    }
}
```

- The method always returns immediately after the first roll.
- This is wrong if that roll isn't a 7; we need to keep rolling.

Returning at the right time

```
// Draws 10 lotto numbers; returns true if one is 7.
public static boolean seven(Random rand) {
    for (int i = 1; i <= 10; i++) {
        int num = rand.nextInt(30) + 1;
        System.out.print(num + " ");
        if (num == 7) {    // found lucky 7; can exit now
            return true;
        }
    }
    return false;   // if we get here, there was no 7
}
```

- Returns `true` immediately if 7 is found.
- If 7 isn't found, the loop continues drawing lotto numbers.
- If all ten aren't 7, the loop ends and we return `false`.

`Random`/`while` question

- Write a multiplication tutor program.
  - Use a static method that returns a boolean value.
  - Test multiplication of numbers between 1 and 20.
  - The program stops after too many incorrect answers.

```
Mistakes allowed: 0

14 * 8 = 112
Correct!
5 * 12 = 60
Correct!
8 * 3 = 24
Correct!
5 * 5 = 25
Correct!
20 * 14 = 280
Correct!
19 * 14 = 256
Incorrect; the answer was 266
You solved 5 correctly.
```


Assertions

reading: 5.5

Logical assertions

- **assertion**: A statement that is either true or false.

  Examples:
  - Java was created in 1995.
  - The sky is purple.
  - 23 is a prime number.
  - 10 is greater than 20.
  - x divided by 2 equals 7. *(depends on the value of x)*

- An assertion might be false ("The sky is purple" above), but it is still an assertion because it is a true/false statement.

Reasoning about assertions

- Suppose you have the following code:

```
if (x > 3) {
    // Point A
    x--;
} else {
    // Point B
    x++;
    // Point C
}
// Point D
```

- What do you know about `x`'s value at the three points?
  - Is `x > 3`? Always? Sometimes? Never?

Assertions in code

- We can make assertions about our code and ask whether they are true at various points in the code.
  - Valid answers are ALWAYS, NEVER, or SOMETIMES.

```
System.out.print("Type a nonnegative number: ");
double number = console.nextDouble();
// Point A: is number < 0.0 here?          (SOMETIMES)

while (number < 0.0) {
    // Point B: is number < 0.0 here?      (ALWAYS)
    System.out.print("Negative; try again: ");

    number = console.nextDouble();
    // Point C: is number < 0.0 here?      (SOMETIMES)
}

// Point D: is number < 0.0 here?          (NEVER)
```


Reasoning about assertions

- Right after a variable is initialized, its value is known:

```
int x = 3;
// is x > 0?  ALWAYS
```

- In general you know nothing about parameters' values:

```
public static void mystery(int a, int b) {
// is a == 10?  SOMETIMES
```

- But inside an `if`, `while`, etc., you may know something:

```
public static void mystery(int a, int b) {
    if (a < 0) {
        // is a == 10?  NEVER
        ...
    }
}
```


Assertions and loops

- At the start of a loop's body, the loop's test must be `true`:

```
while (y < 10) {
    // is y < 10?  ALWAYS
    ...
}
```

- After a loop, the loop's test must be `false`:

```
while (y < 10) {
    ...
}
// is y < 10?  NEVER
```

- Inside a loop's body, the loop's test may become `false`:

```
while (y < 10) {
    y++;
    // is y < 10?  SOMETIMES
}
```


"Sometimes"

- Things that cause a variable's value to be unknown (often leads to "sometimes" answers):
  - reading from a `Scanner`
  - reading a number from a `Random` object
  - a parameter's initial value to a method

- If you can reach a part of the program both with the answer being "yes" and the answer being "no", then the correct answer is "sometimes".
  - If you're unsure, "Sometimes" is a good guess.

hi

Assertion example 1

```
public static void mystery(int x, int y) {
    int z = 0;

    // Point A
    while (x >= y) {
        // Point B
        x = x - y;
        z++;
        if (x != y) {
            // Point C
            z = z * 2;
        }

        // Point D
    }

    // Point E
    System.out.println(z);
}
```

Which of the following assertions are true at which point(s) in the code? Choose ALWAYS, NEVER, or SOMETIMES.

	x < y	x == y	z == 0
Point A	SOMETIMES	SOMETIMES	ALWAYS
Point B	NEVER	SOMETIMES	SOMETIMES
Point C	SOMETIMES	NEVER	NEVER
Point D	SOMETIMES	SOMETIMES	NEVER
Point E	ALWAYS	NEVER	SOMETIMES

Assertion example 2

```
public static int mystery(Scanner console) {
    int prev = 0;
    int count = 0;
    int next = console.nextInt();

    // Point A
    while (next != 0) {
        // Point B
        if (next == prev) {
            // Point C
            count++;
        }
        prev = next;
        next = console.nextInt();
        // Point D
    }
    // Point E
    return count;
}
```

Which of the following assertions are true at which point(s) in the code? Choose ALWAYS, NEVER, or SOMETIMES.

	next == 0	prev == 0	next == prev
Point A	SOMETIMES	ALWAYS	SOMETIMES
Point B	NEVER	SOMETIMES	SOMETIMES
Point C	NEVER	NEVER	ALWAYS
Point D	SOMETIMES	NEVER	SOMETIMES
Point E	ALWAYS	SOMETIMES	SOMETIMES

Assertion example 3

```
// Assumes y >= 0, and returns x^y
public static int pow(int x, int y) {
    int prod = 1;

    // Point A
    while (y > 0) {
        // Point B
        if (y % 2 == 0) {
            // Point C
            x = x * x;
            y = y / 2;
            // Point D
        } else {
            // Point E
            prod = prod * x;
            y--;
            // Point F
        }
    }
    // Point G
    return prod;
}
```

Which of the following assertions are true at which point(s) in the code? Choose ALWAYS, NEVER, or SOMETIMES.

	y > 0	y % 2 == 0
Point A	SOMETIMES	SOMETIMES
Point B	ALWAYS	SOMETIMES
Point C	ALWAYS	ALWAYS
Point D	ALWAYS	SOMETIMES
Point E	ALWAYS	NEVER
Point F	SOMETIMES	ALWAYS
Point G	NEVER	ALWAYS

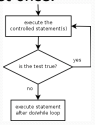
`while` loop variations

reading: 5.1, Appendix D

The `do/while` loop

- **`do/while` loop**: Performs its test at the *end* of each repetition.
  - Guarantees that the loop's `{}` body will run at least once.

```
do {
    statement(s);
} while (test);
```




```
// Example: prompt until correct password is typed
String phrase;
do {
    System.out.print("Type your password: ");
    phrase = console.next();
} while (!phrase.equals("abracadabra"));
```


`do/while` question

- Modify the previous `Dice` program to use `do/while`.

```
2 + 4 = 6
3 + 5 = 8
5 + 6 = 11
1 + 1 = 2
4 + 3 = 7
You won after 5 tries!
```

- Is `do/while` a good fit for our past `Sentinel` program?

bye

hi

do/while answer

```
// Rolls two dice until a sum of 7 is reached.
import java.util.*;

public class Dice {
    public static void main(String[] args) {
        Random rand = new Random();
        int tries = 0;
        int sum;

        do {
            int roll1 = rand.nextInt(6) + 1;   // one roll
            int roll2 = rand.nextInt(6) + 1;
            sum = roll1 + roll2;
            System.out.println(roll1 + " + " + roll2 + " = " + sum);
            tries++;
        } while (sum != 7);

        System.out.println("You won after " + tries + " tries!");
    }
}
```


break

- **`break` statement**: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - The loop's test is often changed to `true` ("always repeat").

```
while (true) {
    statement(s);
    if (test) {
        break;
    }
    statement(s);
}
```

  - `break` is considered to be bad style by some programmers.

Sentinel loop with break

```
Scanner console = new Scanner(System.in);
int sum = 0;
while (true) {
    System.out.print("Enter a number (-1 to quit): ");
    int number = console.nextInt();
    if (number == -1) {      // don't add -1 to sum
        break;
    }
    sum = sum + number;      // number != -1 here
}
System.out.println("The total was " + sum);
```


bye