Building Java Programs

Chapter 5
Lecture 5-2: Random Numbers; Type `boolean`

reading: 5.1, 5.3, 5.6

Random numbers

reading: 5.1

The `Random` class

- A `Random` object generates pseudo-random numbers.
  - Class `Random` is found in the `java.util` package.

```
import java.util.*;
```

Method name	Description
`nextInt()`	returns a random integer
`nextInt(`**`max`**`)`	returns a random integer in the range [0, *max*) in other words, 0 to *max*-1 inclusive
`nextDouble()`	returns a random real number in the range [0.0, 1.0)

  - Example:

```
Random rand = new Random();
int randomNumber = rand.nextInt(10);   // 0-9
```


Generating random numbers

- Common usage: to get a random number from 1 to *N*

```
int n = rand.nextInt(20) + 1;   // 1-20 inclusive
```

- To get a number in arbitrary range [*min*, *max*] inclusive:

  name`.nextInt(`**size of range**`) + `**min**

  - where (**size of range**) is (**max** - **min** + 1)

  - Example: A random integer between 4 and 10 inclusive:

```
int n = rand.nextInt(7) + 4;
```


`Random` questions

- Given the following declaration, how would you get:

```
Random rand = new Random();
```

  - A random number between 1 and 47 inclusive?

```
int random1 = rand.nextInt(47) + 1;
```

  - A random number between 23 and 30 inclusive?

```
int random2 = rand.nextInt(8) + 23;
```

  - A random even number between 4 and 12 inclusive?

```
int random3 = rand.nextInt(5) * 2 + 4;
```


`Random` and other types

- `nextDouble` method returns a `double` between 0.0 - 1.0
  - Example: Get a random GPA value between 1.5 and 4.0:

```
double randomGpa = rand.nextDouble() * 2.5 + 1.5;
```

- Any set of possible values can be mapped to integers
  - code to randomly play Rock-Paper-Scissors:

```
int r = rand.nextInt(3);
if (r == 0) {
    System.out.println("Rock");
} else if (r == 1) {
    System.out.println("Paper");
} else {  // r == 2
    System.out.println("Scissors");
}
```

Random question

- Write a program that simulates rolling of two 6-sided dice until their combined result comes up as 7.

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


Random 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 = 0;
        while (sum != 7) {
            // roll the dice once
            int roll1 = rand.nextInt(6) + 1;
            int roll2 = rand.nextInt(6) + 1;
            sum = roll1 + roll2;
            System.out.println(roll1 + " + " + roll2 + " = " + sum);
            tries++;
        }

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


Type boolean

reading: 5.3

Type boolean

- **boolean**: A logical type whose values are `true` and `false`.
  - A logical **test** is actually a `boolean` expression.
  - It is legal to:
    - create a `boolean` variable
    - pass a `boolean` value as a parameter
    - return a `boolean` value from methods
    - call a method that returns a `boolean` and use it as a test

```
boolean minor    = (age < 21);
boolean isProf   = name.contains("Prof");
boolean lovesCSE = true;

// allow only CSE-loving students over 21
if (minor || isProf || !lovesCSE) {
    System.out.println("Can't enter the club!");
}
```


Using boolean

- Why is type `boolean` useful?
  - Can capture a complex logical test result and use it later
  - Can write a method that does a complex test and returns it
  - Can pass around the result of a logical test (as param/return)
  - Makes code more readable

```
boolean goodAge    = age >= 12 && age < 29;
boolean goodHeight = height >= 78 && height < 84;
boolean rich       = salary >= 100000.0;
if ((goodAge && goodHeight) || rich) {
    System.out.println("Okay, let's go out!");
} else {
    System.out.println("It's not you, it's me...");
}
```


Methods that return boolean

- Methods can return `boolean` values.
  - A call to such a method can be a loop or `if`'s ***<test>***.

```
Scanner console = new Scanner(System.in);
System.out.print("Type your name: ");
String line = console.nextLine();

if (line.startsWith("Dr.")) {
    System.out.println("What's up, doc?");
} else if (line.endsWith(", Esq.")) {
    System.out.println("And I am Ted 'Theodore' Logan!");
}
```

Returning boolean

```
public static boolean isPrime(int n) {
    int factors = 0;
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            factors++;
        }
    }
    if (factors == 2) {
        return true;
    } else {
        return false;
    }
}
```

- Calls to methods returning `boolean` can be used as tests:

```
if (isPrime(57)) {
    ...
}
```


boolean return question

- Write a method `hasVowel` that returns whether or not a String contains a vowel.
  - `hasVowel("oink")` returns `true`
  - `hasVowel("gypsy")` returns `false`
- Write a method `isAllVowels` that returns whether or not a String contains all vowels.
  - `isAllVowels("OUI")` returns `true`
  - `isAllVowels("oink")` returns `false`
  - Write method `isVowel` to complete this two methods.

"Boolean Zen", part 1

- Students new to `boolean` often test if a result is `true`:

```
if (isPrime(57) == true) {     // bad
    ...
}
```

- But this is unnecessary and redundant. Preferred:

```
if (isPrime(57)) {             // good
    ...
}
```

- A similar pattern can be used for a `false` test:

```
if (isPrime(57) == false) {    // bad
if (!isPrime(57)) {            // good
```


"Boolean Zen", part 2

- Methods that return `boolean` often have an `if/else` that returns `true` or `false`:

```
public static boolean bothOdd(int n1, int n2) {
    if (n1 % 2 != 0 && n2 % 2 != 0) {
        return true;
    } else {
        return false;
    }
}
```

- But the code above is unnecessarily verbose.

Solution w/ boolean variable

- We could store the result of the logical test.

```
public static boolean bothOdd(int n1, int n2) {
    boolean test = (n1 % 2 != 0 && n2 % 2 != 0);
    if (test) {   // test == true
        return true;
    } else {      // test == false
        return false;
    }
}
```

- Notice: Whatever `test` is, we want to return that.
  - If `test` is `true` , we want to return `true`.
  - If `test` is `false`, we want to return `false`.

Solution w/ "Boolean Zen"

- Observation: The `if/else` is unnecessary.
  - The variable `test` stores a `boolean` value; its value is exactly what you want to return. So return that!

```
public static boolean bothOdd(int n1, int n2) {
    boolean test = (n1 % 2 != 0 && n2 % 2 != 0);
    return test;
}
```

- An even shorter version:
  - We don't even need the variable `test`. We can just perform the test and return its result in one step.

```
public static boolean bothOdd(int n1, int n2) {
    return (n1 % 2 != 0 && n2 % 2 != 0);
}
```

hi

"Boolean Zen" template

- Replace

```
public static boolean name(parameters) {
    if (test) {
        return true;
    } else {
        return false;
    }
}
```

- with

```
public static boolean name(parameters) {
    return test;
}
```


Improved `isPrime` method

- The following version utilizes Boolean Zen:

```
public static boolean isPrime(int n) {
    int factors = 0;
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            factors++;
        }
    }
    return factors == 2;   // if n has 2 factors -> true
}
```


`boolean` Zen question

- Rewrite `isVowel` to use boolean Zen.

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

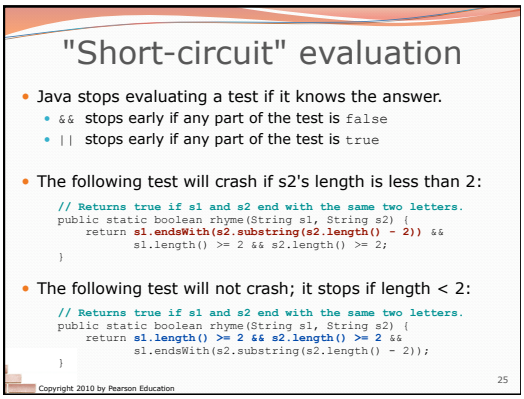
```
// Enlightened version.  I have seen the true way (and false way)
public static boolean isVowel(String s) {
    return s.equalsIgnoreCase("a") || s.equalsIgnoreCase("e") ||
           s.equalsIgnoreCase("i") || s.equalsIgnoreCase("o") ||
           s.equalsIgnoreCase("u");
}


// 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);
}
```


bye

hi



"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));
}
```


bye