CS101

Java Programming

Winter 2013

Thursdays 10:00-12:00noon

RF Academy

Announcements

- Questions??

Boolean return questions

- `hasAnOddDigit` : returns `true` if any digit of an integer is odd.
  - `hasAnOddDigit(4822116)` returns `true`
  - `hasAnOddDigit(2448)` returns `false`

- `allDigitsOdd` : returns `true` if every digit of an integer is odd.
  - `allDigitsOdd(135319)` returns `true`
  - `allDigitsOdd(9174529)` returns `false`

- `isAllVowels` : returns `true` if every char in a `String` is a vowel.
  - `isAllVowels("eIeIo")` returns `true`
  - `isAllVowels("oink")` returns `false`

Boolean return answers

```
public static boolean hasAnOddDigit(int n) {
    while (n != 0) {          // while n still has digits
        if (n % 2 != 0) {     // check whether last digit is odd
            return true;
        }
        n = n / 10;           // remove last digit
    }
    return false;
}

public static boolean allDigitsOdd(int n) {
    while (n != 0) {
        if (n % 2 == 0) {     // check whether last digit is even
            return false;
        }
        n = n / 10;
    }
    return true;
}

public static boolean isAllVowels(String s) {
    for (int i = 0; i < s.length(); i++) {
        String letter = s.substring(i, i + 1);
        if (!isVowel(letter)) {  // isVowel defined a bit later
            return false;
        }
    }
    return true;
}
```

Short-circuit Evaluation

- Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.
  - If the first (left) operand associated with an `||` is `true`, the expression is `true`.
  - If the first (left) operand associated with an `&&` is `false`, the expression is `false`.

- This is called *short-circuit* or *lazy* evaluation.

Short-circuit Evaluation, cont.

- Short-circuit evaluation is not only efficient, sometimes it is essential!

- A run-time error can result, for example, from an attempt to divide by zero.

```
if ((number != 0) && (sum/number > 5)) { ...
```

- *Complete evaluation* can be achieved by substituting `&` for `&&` or `|` for `||`.

De Morgan's Law

- **De Morgan's Law**:
  Rules used to *negate* or *reverse* boolean expressions.
  - Useful when you want the opposite of a known boolean 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) {`
	`  ...`
	`}`

De Morgan Mini-exercises

- For the following statements, negate the test in the “if”:

Original Code	Negated Code

`if (0<=x && x<=10) ) {`
`  ...`
`}`
	`if (x<0 || x>10) {`
	`  ...`
	`}`

`if (a<10 || b<10) ) {`
`  ...`
`}`
	`if (a>=10 && b>=10) {`
	`  ...`
	`}`

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);
}
```

Announcements

- Read sections 5.4-5.6
- Questions??

Invalid user input

- Recall: When the token doesn't match the type the `Scanner` tries to read, the program crashes.

  Example:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you? ");
int age = console.nextInt();
```

  Output (user's input is underlined):

```
What is your age? Timmy
java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        ...
```

Testing for valid user input

- The `Scanner` class has methods that can be used to "look ahead" to test whether the upcoming input token is of a given type:

Method	Description
`hasNext()`	Whether the next token can be read as a `String` *(always true for console input)*
`hasNextInt()`	Whether the next token can be read as an `int`
`hasNextDouble()`	Whether the next token can be read as a `double`
`hasNextLine()`	Whether the next line of input can be read as a `String` *(always true for console input)*

- Each method waits for the user to type input and press Enter, then reports a `true` or `false` answer based on what was typed.
  - The `hasNext` and `hasNextLine` methods are not useful until we learn how to read input from files in Chapter 6.

Scanner condition example

- The `hasNext` methods are useful for testing whether the user typed the kind of token we wanted.
  - This way we can avoid potential exceptions from input mismatches.
  - Example:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you? ");

if (console.hasNextInt()) {
    int age = console.nextInt();  // will not throw an exception
    System.out.println("Retire in " + (65 - age) + " years.");
} else if (console.hasNextDouble()) {
    System.out.println("Please use a whole number for your age!");
    console.nextDouble();  // consume the bad data
} else {
    System.out.println("You did not type a number.");
    console.next();   // consume the bad data as a string
}
```

Infinite Loops

- A loop which repeats without ever ending is called an *infinite loop.*

- If the controlling boolean expression never becomes false, a `while` loop or a `do-while` loop will repeat without ending.

- Use Ctrl-C to stop a program caught in an infinite loop [Eclipse also has a terminate button on the console window (it's the red square)]

`break` Statement

- `break` statement: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - Such loops are often called *"forever" loops* because their header's boolean test is often changed to a trivial `true`.

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

  - `break` is considered to be bad style by some programmers.
  - Not necessary for stuff we do in this class. *Do not use it on CS101 homework! Rather re-write the loop with a different structure.*

Sentinel loop with `break`

- A working sentinel loop solution using `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);
```

Thoughts on break

- Literal meaning is go to after the loop *right now*
  - Needed to solve certain types of problems – just not any problems you will encounter in CS101
- Affects assertions (which we will talk about next)
  - No longer know whether the loop test is false right after the loop
- Can also use `return` anywhere in a method
  - Returns "*right now*" to point of the call