CS 112 Introduction to Programming

Conditional Statements
Boolean Expressions and Methods

Yang (Richard) Yang
Computer Science Department
Yale University
308A Watson, Phone: 432-6400
Email: yry@cs.yale.edu
Admin.

- Walkthrough of PS4
  - 8 pm at DL 220 on Monday
  - 6 pm at DL 220 on Tuesday

- Office hour scheduling
Recap: **Scanner**

- **Some Scanner methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nextInt()</code></td>
<td>Returns an <code>int</code> from source</td>
</tr>
<tr>
<td><code>nextDouble()</code></td>
<td>Returns a <code>double</code> from source</td>
</tr>
<tr>
<td><code>next()</code></td>
<td>Returns a one-word <code>String</code> from source</td>
</tr>
<tr>
<td><code>nextLine()</code></td>
<td>Returns a one-line <code>String</code> from source</td>
</tr>
</tbody>
</table>

- **Java uses objects to remember the source of a scanner**

```java
Scanner console = new Scanner(System.in);
console.nextInt();
```
Recap: Scanning Details

- `nextInt()`, `nextDouble()`, `next()` are token based scanning methods.

- `nextLine()` collects any input character into a string until the first new line and discards the new line.

- **When `nextInt()`, `nextDouble()` parses a token, but cannot parse it correctly, the method throws an exception.**

- A robust program will check condition before invoking `nextInt()` or `nextDouble()`.
Conditional statements, also called decision statements, decide whether or not to execute a particular sequence of statements.
The if Statement

Example:
```java
int nA = 0;
for (int i = 0; i < 100; i++) {
    double grade = console.nextDouble();
    if (grade >= 90) {
        System.out.println("Welcome to Yale!");
        nA ++; // This is called accumulative sum
    }
    nA ++;
}
```
Recall: Basic Boolean Expression

A basic Boolean expression is to compare two values using a *relational operator*:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
<td>1 + 1 == 2</td>
<td>true</td>
</tr>
<tr>
<td>!=</td>
<td>does not equal</td>
<td>3.2 != 2.5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>10 &lt; 5</td>
<td>false</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>10 &gt; 5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>126 &lt;= 100</td>
<td>false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>5.0 &gt;= 5.0</td>
<td>true</td>
</tr>
</tbody>
</table>

Note the difference between the equality operator (==) and the assignment operator (=).
Example: Basic Boolean Expression

```java
public class Flip {
    public static void main(String[] args) {
        if (Math.random() < 0.5) System.out.println("Heads");
        else System.out.println("Tails");
    }
}
```

Flip.java
Outline: Conditional Program Flow Control

- Our “journey” of introducing conditionals
  - simple if (a <comp> b)
  - nested if/else
Motivation: Chaos Game

- Play on equilateral triangle, with vertices R (node 0), G (node 1), B (node 2)
  - Start at R
  - Repeat N times
    - Pick a random vertex
    - Move halfway between current point and vertex
    - Draw a point in color of chosen vertex

Chaos.java
Motivation: Chaos Game

% java Chaos

Sierpinski triangle
Motivation: Chaos Game

```java
public static void main (String[] args) {
    for (int i = 0; i < 1000; i++) {
        int rand = getRand(0, 2);
        if (rand == 0) {
            //
            
        } else if (rand == 1) {
            // ...
            
        } else if (rand == 2) {
            // ...
            
        }
    }
}
```

Q: How many comparisons in each round (iteration)?
public static void main (String[] args) {
    ...
    for (int i = 0; i < 1000; i++) {
        int rand = getRand(0, 2);
        if (rand == 0) {
            //
        } else if (rand == 1) {
            // ...
        } else if (rand == 2) {
            // ...
        }
    }
}

Q: Average # of comparisons per iteration?
Benefit of nested comparison: reduce # comparisons
How is this code for grading based on percentiles?

Scanner console = new Scanner(System.in);
System.out.print("What percentile? ");
int percent = console.nextInt();
if (percent >= 90) {
    System.out.println("You got an A!");
} else
if (percent >= 80) {
    System.out.println("You got a B!");
} else
if (percent >= 70) {
    System.out.println("You got a C!");
} else
if (percent >= 60) {
    System.out.println("You got a D!");
} else
if (percent < 60) {
    System.out.println("You got an F!");
}

...
Summary: Why nested if/else

- **Mutually exclusive test conditions**
  
  => Reduces # of comparisons

- **Non-mutual exclusive test conditions**
  
  => Achieves mutual exclusion

```
if (percent >= 90) {
    // assigns A
} else
    if (percent >= 80) {
        // assigns B
    } else
        if (percent >= 70) {
            // assigns C
        } else
            if (percent >= 60) {
                // D
            } else {
                // F
            }
```
Exercise: Barnsley Game

- Play Chaos game with different rules

<table>
<thead>
<tr>
<th>probability</th>
<th>new x</th>
<th>new y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2%</td>
<td>.50</td>
<td>.27y</td>
</tr>
<tr>
<td>15%</td>
<td>-.14x + .26y + .57</td>
<td>.25x + .22y - .04</td>
</tr>
<tr>
<td>13%</td>
<td>.17x - .21y + .41</td>
<td>.22x + .18y + .09</td>
</tr>
<tr>
<td>70%</td>
<td>.78x + .03y + .11</td>
<td>-.03x + .74y + .27</td>
</tr>
</tbody>
</table>

Barnsley.java
Exercise: Barnsley Game

public static void main(String[] args) {
    StdDraw.setPenRadius(0.002);
    StdDraw.setPenColor(Color.GREEN);

    int T = 40000;
    double x = 0.0, y = 0.0;

    for (int t = 0; t < T; t++) {
        double rand = Math.random(); // 0 - 1.0

        if (rand <= 0.7) { // 0 - 0.7: 70%
            x = .78 * x + .03 * y + .11;
            y = -.03 * x + .74 * y + .27;
        } else if (rand <= 0.7 + 0.15) { // 0.7 - 0.7+0.15: 15%
            x = -.14 * x + .26 * y + .57;
            y = .25 * x + .22 * y - .04;
        } else if (rand < 0.7 + 0.15 + 0.13) { // 0.7+0.15 - 0.7+0.15+0.13: %13
            x = .17 * x - .21 * y + .41;
            y = .22 * x + .18 * y + .09;
        } else { // 2%
            x = 0.5;
            y = .27 * y;
        }
    }
    StdDraw.point(x, y);
}
} // end of main
Exercise: Barnsley Game

Questions to think about

❖ What does computation tell us about nature?
❖ What does nature tell us about computation?
Outline: Conditional Program Flow Control

- Our “journey” of introducing conditionals
  - simple if \((a < \text{comp} b)\)
  - nested if/else

  Complexity of nested if/else:
  all path must return; mismatched else
Matching Nested if Statements

- Nested if statements may have a matching problem

```java
if (temperature < 50)
if (temperature < 100)
    System.out.println("Cool!");
else
    System.out.println("Hot!");
```
Nested if w/ Ambiguity

If (temperature < 50)
  If (temperature < 100)
    System.out.println("Cool!");
  else
    System.out.println("Hot!");

Give a value of temperature to produce different results
Nested if Statements

- Rule: an else clause is matched to the last unmatched if (no matter what the indentation implies)

- If you find such statements confusing, avoid writing such structures and always use block statements ({{}}) to make the structure clear.

```java
if (temperature < 50)
  if (temperature < 100)
    System.out.println(“Cool!”);
else
  System.out.println(“Hot!”);

if (temperature < 50)
  if (temperature < 100)
    System.out.println(“Cool!”);
else
  System.out.println(“Hot!”);
```
A method with a return requires that all paths through the code must reach a return statement.

The compiler analyzes this by considering the syntax only!
static int myMethod() {
    if (test1) {
        statement(s);
        return x;
    } else if (test2) {
        statement(s);
        return y;
    } else {
        statement(s);
        return z;
    }
} // end of method
if/else with return

```java
static int myMethod() {
    if (test1) {
        statement(s);
        return x;
    } else if (test2) {
        statement(s);
        return y;
    } else {
        statement(s);
        return z;
    }
} // end of method
// ERROR: missing return in a path
```
if/else with return

```java
public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    // Error: not all paths return a value
}

public static int max(int a, int b) {
    if (a > b) {
        return a;
    } else if (a <= b) {
        return b;
    }
    // Error: syntax analysis missing a path
}

public static int max(int a, int b) {
    if (a > b) {
        return a;
    } else if (a <= b) {
        return b;
    }
    // OK
}

public static int max(int a, int b) {
    int myMax = a; // default
    if (b > a) {
        myMax = b;
    }
    return myMax; // OK
}

public static int max(int a, int b) {
    if (b > a) {
        return b;
    }
    return a; // OK
}
```
Outline: Conditional Program Flow Control

- Our “journey” of introducing conditionals
  - simple if (a <comp> b)
  - nested if/else

  Complexity of nested if/else:
  all path must return; mismatched else

  <condition> combining multiple <comp>;
Motivation: Testing Containment

- We may need to check multiple conditions

Test if point \((x, y)\) is in the rectangle?

\[(0 \leq x \leq W) \text{ and } (0 \leq y \leq H)\]

SYNTAX ERROR

\[(0 \leq x) \text{ and } (x \leq W) \text{ and } (0 \leq y) \text{ and } (y \leq H)\]
Logical Operators

- Tests can be combined using **logical operators**:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>(2 == 3) &amp;&amp; (-1 &lt; 5)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>!(2 == 3)</td>
<td>true</td>
</tr>
</tbody>
</table>

- "Truth tables" for each, used with logical values $p$ and $q$:

| $p$   | $q$   | $p$ && $q$ | $p$ || $q$ |
|-------|-------|------------|-----------|
| true  | true  | true       | true      |
| true  | false | false      | true      |
| false | true  | false      | true      |
| false | false | false      | false     |

<table>
<thead>
<tr>
<th>$p$</th>
<th>!$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
Extension: Testing Containment

- We may need to check multiple conditions

Test if dark rectangle is in the bigger rectangle?
Exercise

- Write a method to compute the number of days in a month

```java
public static int daysInMonth(int year, int month)
```
Applying Boolean Exp. in daysInMonth

- Use a logical OR to combine the cases

```java
if (m == 1 || m == 3 || m == 5 ||
    m == 7 || m == 8 || m == 10 || m == 12)
    numDays = 31;
```
Applying Boolean Exp. in daysInMonth

- Implement leap year condition: “... most years that are evenly divisible by 4 are leap years; However, there are some exceptions to this rule: Years that are evenly divisible by 100 are not leap years, unless they are also evenly divisible by 400”

leap year

or

year % 400 == 0

and

year % 4 == 0

year % 100 != 0
Testing Leap Year

leap year

or

eyear % 400 == 0

and

year % 4 == 0

year % 100 != 0

y % 400 == 0 || (y % 100 != 0 && y % 4 == 0)
Outline: Conditional Program Flow Control

- Our “journey” of introducing conditionals

  simple if (a <comp> b)

  nested if/else

  Complexity of nested if/else:
  all path must return; mismatched else

  <condition> combining multiple <comp>;

  Boolean variables/expressions/methods
Reuse Testing for Leap Year

How do we define a method to reuse the ability to test if a year is a leap year?

```java
?? isLeapYear(int year);
```
Recall: **boolean Type**

- **boolean**: A primitive type whose values are **true** and **false**.
  - Like other types, it is legal to:
    - create a **boolean** variable and assign it values
    - pass a **boolean** value as a parameter
    - return **boolean** value as a method
    - call a method that returns a **boolean** and use it as a test
**boolean Expressions**

- Similar to arithmetic expressions, except that
  - the operands are Boolean values, Boolean variables, or a test using a relational operator (e.g., <, >).
  - the operators are || && !

- Example
  ```java
  boolean lovesCS = true;
  boolean student = age < 21;

  // allow only CS-loving students over 21
  if (student && lovesCS)
    System.out.println("Pass");

  // an alternative
  if (age < 21 && lovesCS)
    System.out.println("Pass");
  ```
boolean Expressions: Example

Boolean goodAge = age >= 18 && 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...");
}
Mixed Arithmetic, Relation, Logical, and Assignment Expression

Example:
```java
boolean mystery = 5 * 7 >= 3 + 5 * (7 - 1) && 7 <= 11;
```

Precedence ordering of boolean expression
- Arithmetic operators
- Relations operators (==, !=, <, >, <=, >=)
  - Note that equality and relational operators cannot be chained (e.g., 1 < x < 3 is invalid)
- NOT (!)
- AND (&&)
- OR (||)
- Assignment operators (=)
Example

boolean mystery = 5 * 7 >= 3 + 5 * (7 - 1) && 7 <= 11;

5 * 7 >= 3 + 5 * 6 && 7 <= 11
35 >= 3 + 30 && 7 <= 11
35 >= 33 && 7 <= 11
true && true
true
English vs boolean

- **OR vs Exclusive OR**
  - I’ll either watch TV or go to gym: Exclusive OR
  - `watchTV || gotoGym` can be both true

- **x is between 1 and 10**
  - `1 <= x <= 10`
  - `1 <= x && x <= 10`

- **x is either 1 or 2 or 3**
  - `x == 1 || 2 || 3`
  - `x ==1 || x == 2 || x == 3`
Logical AND/OR Evaluation

- **Java uses shortcircircuit when evaluating && and ||**
  - **a && b shortcircircuit:**
    - if a is false, b is not evaluated
  ```java
  if ((count != 0) && (total / count > AVG_THRESHOLD)) {
    // ...
  }
  ```
  - **a || b shortcircircuit:**
    - if a is true, b is not evaluated