CS 112 Introduction to Programming

Nested if/else;
Switch and Conditional Operator Shortcuts;
Strings Processing

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson, Phone: 432-6400
Email: yry@cs.yale.edu
PS4 questions

PS5 posted on the Assignments page

- Monte Carlo is fully doable
- TaskMan needs some background on Arrays which we cover on Wednesday

Supporting materials of Exam 1 are on the Schedule page
Recap: Boolean Expressions, Type

- Program flow of control uses Boolean expressions:
  - The operands are relational tests, Boolean values, and Boolean variables.
  - The operators are logical operators || && !

- The boolean type:
  - Use as variables, method return
  - Boolean “zen”
Recap: Nested Comparison (else if)

```java
for (int i = 0; i < 1000; i++) {
    int rand = (int) (Math.random() * 3);
    if (rand == 0) {
        // do 0
    }
    if (rand == 1) {
        // do 1
    }
    if (rand == 2) {
        // do 2
    }
}
```

Benefit of nested comparison: reduce # comparisons

```java
for (int i = 0; i < 1000; i++) {
    int rand = (int) (Math.random() * 3);
    if (rand == 0) {
        // do 0
    }
    else if (rand == 1) {
        // do 1
    }
    else if (rand == 2) {
        // do not 0 or 1
    }
}
```

5/3 = 1/3 * (1 + 2 + 2) comp.
A Common Use of Nested if/else: “Stairstep Testing”
Example: Grading Curve

```java
if (percent >= 90) {
  grade = A;
}

if (percent >= 80) {
  grade = B;
}

if (percent >= 70) {
  grade = C;
}

if (percent >= 60) {
  grade = D;
}

if (percent < 60) {
  grade = F;
}
```
Example: Grading Curve (Design 1)

```java
if (percent >= 90) {
    grade = A;
}

if (percent >= 80 && percent < 90) {
    grade = B;
}

if (percent >= 70 && percent < 80) {
    grade = C;
}

if (percent >= 60 && percent < 70) {
    grade = D;
}

if (percent < 60) {
    grade = F;
}

Exclusive if conditions
```
Example: Grading Curve (Design 2)

```
if (percent >= 90) {
    grade = A;
}
else if (percent >= 80) {
    grade = B;
}
else if (percent >= 70) {
    grade = C;
}
else if (percent >= 60) {
    grade = D;
}
else if (percent < 60) {
    grade = F;
}
```

Nested comparison to achieve mutual exclusion; high-to-low.
Nested if/else as Set Refinement

```java
if (percent >= 90) {
    // assigns A
} else
if (percent >= 80) {
    // assigns B
} else
if (percent >= 70) {
    // assigns C
} else
if (percent >= 60) {
    // D
} else {
    // F
}
```
Example: Grading Curve (Design 3: Low to High)

```java
if (percent < 60) {
    grade = F;
}
else if (percent < 70) {
    grade = D;
}
else if (percent < 80) {
    grade = C;
}
else if (percent < 90) {
    grade = B;
}
else {
    grade = A;
}
```
Example: Grading Curve using nested else/if

```java
if (percent >= 90) {
    grade = A;
} else if (percent >= 80) {
    grade = B;
} else if (percent >= 70) {
    grade = C;
} else if (percent >= 60) {
    grade = D;
} else {
    grade = F;
}
```

```java
if (percent < 60) {
    grade = F;
} else if (percent < 70) {
    grade = D;
} else if (percent < 80) {
    grade = C;
} else if (percent < 90) {
    grade = B;
} else {
    grade = A;
}
```

High-to-low or Low-to-high is a personal programming style.
Exercise: Chaos Game 2

- Play a chaos game with these 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>

- Basic idea
  - Generate random numbers in [0, 1.0]
  - Divide 0-1.0 into 4 areas for the 4 cases
  - Check which area each number falls in
Exercise: Chaos Game2

- Play a chaos game with these 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>

- Basic idea
  - Generate random numbers in [0, 1.0]
  - Divide 0-1.0 into 4 areas for the 4 cases
  - Check which area each number falls in
public static void main(String[] args) {
    StdDraw.setPenRadius(0.002);
    StdDraw.setPenColor(Color.GREEN);

    final 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 for
} // end of main
Barnsley Game

Questions to think about

- What does computation tell us about nature?
- What may nature tell us about computation?
Nested if/else is a common structure
  - Mutually exclusive test conditions
    => Reduces # of comparisons
  - Non-mutual exclusive test conditions
    => Achieves mutual exclusion
Outline

- Admin and recap
- Conditionals
  - Logical conditions (Boolean expressions)
    - Basic relational tests
    - Tests using logical operators
    - Boolean type
  - Nested if/else conditional statements
    - Benefits of nested if/else
    - Complexities of nested if/else
      - mismatched else
      - all path must return
Nested if statements may have a matching problem

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

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

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

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

- **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.
**if/else with return**

- 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!
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
```
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
}
```

```java
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
}
```

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

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

```java
public static int max(int a, int b) {
    if (b > a) {
        return b;
    }
    return a; // OK
}
```
Outline

- Admin and recap
- Conditionals
  - Logical conditions (Boolean expressions)
    - Basic relational tests
    - Tests using logical operators
    - Boolean type
  - Nested if/else conditional statements
    - Benefits of nested if/else
    - Complexities of nested if/else
      - mismatched else
      - all path must return
  - Conditional shortcuts
Alternative Testing using switch

- Java `switch` statement is a shortcut allowing clear listing of multiple choices in some settings.

  - `expression` must result in an integral data type, like an integer or char.

  ```java
  switch (expression)
  {
  case value1 :
    statement-list1
    break;
  case value2 :
    statement-list2
    break;
  case value3 :
    statement-list3
    break;
  case ...
  }
  ```

- `switch` and `case` are keywords.
- If `expression` first matches `value2`, control jumps to here.
A switch statement can have an optional `default case` as the last case in the statement.

The default case has no associated value and simply uses the reserved word `default`.

If the default case is present, control will transfer to it if no other case value matches.

If there is no default case, and no other value matches, control falls through to the statement after the switch.
**The switch Statement Design Decision**

- Flow of control jumps to the first matching case, and execution continues ignoring the case statements.

- Often a `break` statement is used at the end of each case's statement list.
  - A `break` statement causes control to transfer to the end of the `switch` statement.

- If a `break` statement is not used, the flow of control will continue into the next case.
Exercises

- Implement DaysInMonth.java using switch (DaysInMonthSwitch.java)

- (Offline) Implement a method to convert a number between 1 and 10 to its English word
Limitation of the `switch` Statement

- The result of the `expression` must be an integral type, e.g.,
  - `case x < 1: // ERROR`

- The value for each case must be a constant expression

```java
switch ( expression ) {
    case value1 :
        statement-list1
    case value2 :
        statement-list2
    case value3 :
        statement-list3
    default: ...
}
```
Shortcut: The Conditional Operator

- Java has a *conditional operator* that evaluates a *boolean expression* to determine which one of the two expressions should be evaluated:

  \[
  \text{condition} \ ? \ \text{expression1} \ : \ \text{expression2}
  \]

- If the *condition* is true, *expression1* is evaluated; if it is false, *expression2* is evaluated; the value of the evaluated expression becomes available.

- For example:

  \[
  \text{larger} = (\text{num1} > \text{num2}) \ ? \ \text{num1} \ : \ \text{num2};
  \]

  - *if* num1 *is greater than* num2, *then* num1 *is assigned to* larger; *otherwise, num2 is assigned to* larger.
Examples

Example 1:
Scanner scan = new Scanner(System.in);
double val = scan.nextDouble();
val = (val < 0) ? -val : val;

Exercise: Use operator on Feb (daysInMonth)

Offline exercise (use right plural):
Scanner scan = new Scanner(System.in);
int dollar = scan.nextInt();
System.out.println("Your amount is "+ dollar +
_________ ? "Dollar." : "Dollars.");
any program you might want to write

- objects
- methods
- classes
- graphics
- sound
- image
- arrays
- conditionals
- loops
- math
- text
- primitive data types
- assignment statements
Outline

- Admin and recap
- Conditionals
- Strings
Discussion

- Functions (methods) one may need to process text (Strings)?
Roadmap: String Processing

- **Access methods**, e.g.,
  - length, substring, charAt, indexOf

- **String generator methods** (string->string), e.g.,
  - toLowerCase, toUpperCase, replace, split (string->strings)

- **Conversion from String**, e.g.,
  - Integer.parseInt, Double.parseDouble

- **Input processing**, e.g.,
  - Scanner.next, Scanner.nextLine

- **Output formatting**: format string
  - System.out.printf, String.format

- **Boolean methods**, e.g.,
  - equals, equalsIgnoreCase, startsWith, endsWith, contains, matches
**Strings**

- **string**: An object storing a sequence of text characters.
  
  Unlike most other objects, a `String` is so common that Java introduces a short cut and you do not need to create String with `new`—to make Strings more similar to native primitive data types.

```java
String name = "text";

String name = expression;
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```
String Internal: Indexes

- A string is a sequence of characters numbered with 0-based indexes:

```java
String name = "R. Kelly";
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>R</td>
<td>.</td>
<td>K</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

- First character's index : 0
- Last character's index : 1 less than the string's length

- The individual characters are values of type `char`
String Access Methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>number of characters in this string</td>
</tr>
<tr>
<td>charAt(index)</td>
<td>character at index</td>
</tr>
<tr>
<td>indexOf(str)</td>
<td>index where the start of the given string appears in this string (-1 if not found)</td>
</tr>
</tbody>
</table>

- These methods are called using the dot notation:

```java
String name = "Dr. Dre";
System.out.println( name.length() );  // 7
```
String Method Examples

// index 012345678901234
String s1 = “Yale University”;
String s2 = “CPSC112, Spring”;
System.out.println( s1.length() ); // 15
System.out.println( s1.indexOf("e") ); // 3
Java strings are immutable. Hence, each of the above will return a new string. With the current string not modified.
String Method Examples

// index 012345678901234
String s1 = "Yale University";
String s2 = "CPSC112, Spring";

System.out.println(s1.length()); // 15
System.out.println(s1.indexOf("e")); // 3
System.out.println(s1.substring(5, 15)); // "University"

String s3 = s2.substring(9, 15);
System.out.println(s3.toLowerCase()); // "spring"
String as Immutable Data Type

- Methods like `substring` and `toLowerCase` build and return a new string, rather than modifying the current string.

```java
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);  // lil bow wow
```

- To modify a variable's value, you must reassign it:

```java
String s = "lil bow wow";
s = s.toUpperCase();
System.out.println(s);  // LIL BOW WOW
```
Parsing a String

- Instead of substring, indexOf, you can also use scanner to parse a string (in units of tokens only).

```java
String s = "Year 2017 is great!";
Scanner scan = new Scanner(s);
String word = scan.next(); // skip Year
int year = scan.nextInt();
```
Exercise

- Given the following string:

```
// index 0123456789012345678901
String book = "Building Java Programs";
```

- How would you extract the word "Building" from book?
- More generally, how to extract the first word of a string?