Midterm Review

COMP 110
Summer II 2012

Peter Lincoln
7/6/2012
Announcements

- Lab 5 and Program 2 were due at 9:45a
Questions?

- **Yesterday:**
  - Program 1 In-Review
  - More About Classes
    - Local v. Instance Variables
    - Methods with Parameters
    - Javadoc
  - In-Class Exercise
    - Arrays
    - Class Basics
Today in COMP 110

- Whirlwind tour of almost everything we have covered so far
Midterm Metadata

- Topics
- Sources
- Scheduling
- Sample
Midterm Topics

- Computer Basics
- Designing Programs
- Primitive Types
- Strings
- Console I/O
- Branching Statements
- JOptionPane
- Loops
- Debugging Techniques
- Arrays
Midterm Material Sources

- Textbook
- Lecture Slides
- Assignments
- In-Class Exercises
- Class Website
  - No External Links
Midterm Structure

- Duration: 55 minutes
- Starting promptly at 9:45a on Monday
- No notes, no textbook, no computer
- Represents 15% of your overall grade
Midterm Rescheduling

- To make up a midterm, you must notify me in advance or have a doctor’s excuse.
Sample Midterm

- Is available on the class website
  - Note that it covers some materials that are not going to be on your midterm
Questions?
Computer Basics
Hardware v. Software

- **Hardware**: Physical Machine
  - **CPU**: The “brain” of your computer
  - **Memory**: Stores data for the computer

- **Software**: Programs that give instructions to the computer
  - **Operating System**
  - **Games**
  - **Java Runtime**
  - **Eclipse**
  - **Your Programs**
Memory

- Measured in Bytes
- 1 Byte = 8 bits
  - Bit is either 0 or 1
- Language of the computer is bits
Programming Languages

- **Your Program**
  - High Level Language
  - Human Readable

- **Compiler**
  - Translates your program

- **Machine Language**
  - Low-Level Language
  - Computer Readable
The Java Programming Language

- Object-Oriented Programming

- Three Main Design Principles
  - Encapsulation: Information Hiding
  - Polymorphism: One method call can cause different actions in different contexts
  - Inheritance: A means of organizing classes into a hierarchy
Designing Programs

What's going on? I'm 10 feet tall and blue!

And my hair has a long braid with a USB connector on the end!

I wonder what would happen if I plugged it in to that computer...

It's like our minds are one! I can speak its language! This is awesome!

Ooo - it wants me to write some object-oriented code...

I had that "Javatart" dream again last night.

You say that like I'm supposed to be jealous.

Algorithms & Pseudocode

- Algorithm: A set of instructions for solving a problem

- Pseudocode: A combination of code and English used to express an algorithm
  - Write before coding to design the algorithm
  - Write after coding to explain the code
Variables

- Used to store data in a program
  - Variable: a storage location in memory
  - Value: the data currently in the variable
  - Identifier: the name of a variable
    - Choose variable names that are meaningful

Declare Variable: \texttt{score}

Assign 5 to \texttt{score}

Main Memory
Variable Declarations

- Syntax:
  - Type variableName1, variableName2, ...;

- Examples
  - int count, score, total;
  - char gradeLetter;
  - double totalCost, ratio;

- int n1; // Not a useful name

- Variable names composed of letters, digits, and underscores
  - Using camelCase is recommended
Keywords

- Keyword: a reserved word with a predefined meaning

- Cannot use keywords for variable names
  - Examples: if, else, return, new
  - A complete list is in the front inside cover of the textbook (5th edition) and is linked on the course website (“6th edition vs. 5th edition”)
Assignment Statements

- Changes a variable’s value

- Syntax:
  - `variable = expression;`

- Examples:
  - `sleepNeeded = 8;`
  - `sleepDesired = sleepNeeded * 2;`
  - `sleepObtained = sleepNeeded / 2;`
Specialized Assignment Operators

- **Unary**: `++`, `--`
- **Binary**: `*=``, `/=`, `%=``, `+=`, `-=`

- Shorthand for commonly performed operations

- **Examples**:  
  - `length = length * 5;`  
    - `length *= 5;`
  - `age = age + 1;`  
    - `age += 1;`  
    - `age++;`
Primitive Types

Source: BC Comic by Johnny Hart
Variable Types

The kind of value the variable can hold

Two main forms of types:

- **Primitive Types**: indecomposable values
  - Names begin with lowercase letters and are keywords
  - Examples: byte, short, int, long, float, double, char, boolean

- **Class Types**: composed of other variables and/or methods
  - Names begin with uppercase letters
  - Examples: Scanner, String
Primitive Types

- **Integers** (byte, short, int, long)
  - 0, -3, 5, 43

- **Floating-Point Numbers** (float, double)
  - 0.5, 12.4863, -4.3

- **Characters** (char)
  - ‘A’, ‘r’, ‘%’, ‘T’

- **Boolean** (boolean)
  - true, false
Variables and Memory

- Different variable types
  - Have different ranges (min, max)
  - Have different precision levels (decimal places)
  - Consume different amounts of memory when declared
  - Full details on the front inside cover of the textbook (or Figure 2.1)

```c
int age;
double length;
char letter;
```
Type Casting

- Usually values of a particular type are put into variables of the same type.
- If not, then in some cases the value will be automatically converted (implicit type cast).

Example:

```java
int age;
age = 10;
double length;
length = age;
```

Auto-cast: `byte → short → int → long → float → double`
Type Casting Operator

- Auto-cast: byte $\rightarrow$ short $\rightarrow$ int $\rightarrow$ long $\rightarrow$ float $\rightarrow$ double

- If a cast in the opposite direction is required, then the casting operator is required

- Illegal (compiler error):
  - myFloat = myDouble;
  - myByte = myInt;
  - myShort = myFloat;

- Legal:
  - myFloat = (float)myDouble;
  - myByte = (byte)myInt;
  - myShort = (short)myFloat;
Strings
Strings

- In programming, a string is a sequence of characters, e.g.
  - “Hello World!”
  - “Enter a whole number from 1 to 99.”

- In Java, the String class handles strings
  - Not a primitive type
  - Each character in the string is char type
String Declaration & Output

- Code Example
  - String animal = "tiger";
  - System.out.println(animal);

- Output
  - tiger
String Concatenation Operator

+ (binary)

Example:

```java
String animal = "tiger";
String sentence = "My favorite animal is the " + animal;
System.out.println(sentence);
```

Output:

My favorite animal is the tiger
String Concatenation Operator

- + (binary)

Example:

- String animal = "tiger";
  String sentences = "My favorite animal is the " + animal + ". What is yours?";
  System.out.println(sentences);

Output:

- My favorite animal is the tiger. What is yours?
String Concatenation Operator

- Concatenating Strings with other types
  - Produces String typed value
  - Example
    - `int n1 = 7, n2 = 13;`  
      String s1 = "I like the number " + n1;  
      String s2 = n2 + " can be unlucky";
String Concatenation Operator

- Concatenating Strings with other types
  - Gotcha: When concatenating a series of non-strings, the left most operand must be a String
    - Else, + means addition, not concatenation
  
  - Wrong: 'H' + 'i' // Yields Number
  
  - Right: "" + 'H' + 'i' // Yields String
## Sample String Methods

<table>
<thead>
<tr>
<th>Invokation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myString.length()</td>
<td>Returns the number of characters in myString</td>
</tr>
<tr>
<td>myString.equals(“a string”)</td>
<td>Returns true if myString and “a string” represent the same string</td>
</tr>
<tr>
<td>myString.toLowerCase()</td>
<td>Returns the lower case version of myString</td>
</tr>
<tr>
<td>myString.trim()</td>
<td>Removes whitespace from the beginning and end of myString</td>
</tr>
<tr>
<td>myString.charAt(5)</td>
<td>Returns the character at index 5</td>
</tr>
<tr>
<td>myString.substring(5)</td>
<td>Returns the substring starting at index 5 (inclusive) and ending at the end of the String</td>
</tr>
<tr>
<td>myString.substring(5, 9)</td>
<td>Returns the substring starting at 5 (inclusive) and ending at 9 (exclusive)</td>
</tr>
</tbody>
</table>
String Indices

- String output = "UNC is Great!";
- int len = output.length(); // 13

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>N</td>
<td>C</td>
<td>i</td>
<td>s</td>
<td></td>
<td></td>
<td>G</td>
<td>r</td>
<td>e</td>
<td>a</td>
<td>t</td>
<td>!</td>
</tr>
</tbody>
</table>

- char ch = output.charAt(7); // 'G'
Substrings

- String output = "UNC is Great!";
- `int len = output.length(); // 13`

<table>
<thead>
<tr>
<th></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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
<td>N</td>
<td>C</td>
<td>i</td>
<td>s</td>
<td></td>
<td>G</td>
<td>r</td>
<td>e</td>
<td>a</td>
<td>t</td>
<td>!</td>
<td></td>
</tr>
</tbody>
</table>

- String sound = output.substring(7, 9);
  // Gr

- Substring yields is from start (inclusive) to end (exclusive)
Substrings

- String output = "UNC is Great!";
- int len = output.length(); // 13

What call would I use to get the word “is” out of this string?
Substrings

- String output = "UNC is Great!";
- int len = output.length(); // 13

What call would I use to get the word “is” out of this string?
- String is = output.substring(4, 6);
# Escape Characters

<table>
<thead>
<tr>
<th>Escape Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>Double-quote, required to store &quot; in a string value</td>
</tr>
<tr>
<td>'</td>
<td>Single-quote, required to store ' in a character value</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>\n</td>
<td>New Line</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\0</td>
<td>NULL character</td>
</tr>
<tr>
<td>\uHHHHH</td>
<td>Unicode character matching HHHH as a hexadecimal value</td>
</tr>
</tbody>
</table>
Console I/O
Console I/O

- Supported by two objects
  - Input: System.in
    - Typically wrapped in a Scanner object
  - Output: System.out
Writing to Console Output

- What is the difference?
  - `System.out.print("This is a string");`
    - Writes the text “This is a string” to the screen
  - `System.out.println("This is a string");`
    - Writes the text “This is a string” to the screen
    - And then advances to the next line

- Equivalent Statement Groups
  - `System.out.println("This is a string");`
  - `System.out.print("This is a string");`
  - `System.out.println();`
Reading from the Keyboard

- For convenience, wrap System.in in a Scanner object
  - Scanner keyboard = new Scanner(System.in);
    int myInt = keyboard.nextInt();
    double myDouble = keyboard.nextDouble();
    String word = keyboard.next();
    String line = keyboard.nextLine();

- Most next...() methods
  - Discard preceding separators
  - Read the value
  - Stops before the following separator

- Gotcha (p. 89 5th ed.): nextLine() reads through the \r\n, but then discards the \r\n.
JOptionPane

Input

Hello World!

Hi There!

OK Cancel
JOptionPane class

- Creates dialog boxes
  - Graphical User Interface (GUI)
  - Alternative to Console I/O

- Requires import statement
  - import javax.swing.*;
JOptionPane Class Methods

- Collect some information from the user
  - String inputString = JOptionPane.showInputDialog("Please enter a string:);
  - showInputDialog only returns Strings
  - To use value as a number or other type, you need to parse it, e.g.
    - int myInt = Integer.parseInt(inputString);
    - double myDouble = Double.parseDouble(inputString);
JOptionPane Class Methods

- Alert the user of some information
  - JOptionPane.showMessageDialog(null, "Your message here");

- Why start with a null?
  - First argument to showMessageDialog is the parent GUI component
    - For the programs you’ve written, there is no parent GUI component, thus use null
System.exit(0);

- A special method
  - Most methods return control of the program to the next statement when they finish
  - System.exit(0); causes the program to stop executing

- Also useful for situations where continued executing is pointless or not desired
Branching Statements
import java.util.*;

public class FlowChart {
    public static void main(String[] args) {
        System.out.println("Give me an integer:");
        Scanner keyboard = new Scanner(System.in);
        int inputInt = keyboard.nextInt();

        if (inputInt > 10) {
            System.out.println("big number");
        } else {
            System.out.println("small number");
        }
    }
}
Branching Statement if/else

Syntax

```java
if (Boolean_Expression)
{
    Then_Statement
}
else
{
    Else_Statement
}
```

If value of `Boolean_Expression` is true, then perform `Then_Statement`, else perform `Else_Statement`
Branching Statement if/else

- Can execute multiple statements per case
  - if (Boolean_Expression)
    
    ```
    Statement_1a
    Statement_1b
    ...
    ```
  
    else
    
    ```
    Statement_2a
    Statement_2b
    ...
    ```

- Recommend using block syntax *always*, even if just using one statement
Branching Statement if/else

- Else statement is optional
  - if (Boolean_Expression)
    {
      Then_Statement
    }

- Only if value of Boolean_Expression is true, then perform Then_Statement
Boolean Expressions

- An expression that evaluates to either true or false

- Examples
  - It is sunny today (true)
  - 10 is larger than 5 (true)
  - Today is Saturday (false)
Don’t Worry about Paradoxes

Well, don’t worry about them until the robot uprising, at least.
Primitives Type: boolean

- Boolean expressions evaluate to a boolean type value
  - Can be either true or false

Example

```java
boolean sunny = true;
boolean cloudy = false;
```
Java Comparison Operators

- When programming, Boolean expressions are often used for comparisons

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Is not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Is strictly greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Is greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Is strictly less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Is less than or equal to</td>
</tr>
</tbody>
</table>

- Examples:
  - myInt <= 6
  - myDouble > 4.0
  - myChar == 'A'
Gotchas with `==`

- `=` vs. `==`
  - `var1 = var2` (assignment statement)
    - Compiler Error!
  - `var1 == var2` (equality comparison)

- String Equality
  - Do not use `==` to compare strings (or any other class type)
    - `string1 == string2`  // BAD
  - Use the equals method instead
    - `string1.equals(string2)`  // GOOD
Boolean Operator: && (AND)

- True if all expressions are true

Example

- Only walk to school if it is between 50 and 75 degrees

```java
if ((temperature > 50) && (temperature < 75)) {
    // walk to school
}
```
Boolean Operator: \(||\) (OR)

- True if **at least one** expression is true

Example

- Walk to school if it is sunny or cloudy
  - `if (isSunny || isCloudy)
    {
      // walk to school
    }`
More about && and ||

- && and || are short-circuiting operators
- Consider these expressions
  - Expression_A && Expression_B
  - Expression_C || Expression_D

- If Expression_A is false, does the value of Expression_B matter?
- If Expression_C is true, does the value of Expression_D matter?
More about && and ||

- && and || are short-circuiting operators
- Consider these expressions
  - Expression_A && Expression_B
  - Expression_C || Expression_D

- If Expression_A is false, then Expression_B will not be evaluated
- If Expression_C is true, then Expression_D will not be evaluated
Short-Circuit Evaluation

- Can be useful to prevent runtime errors, consider
  
  ```java
  if ((count > 0) && ((totalScore / count) >= 80))
  {
    System.out.println("The average is at least 80");
  }
  ```

- If the value of count were zero, then the right expression would have produced an error (divide by zero)
  
  - Short-circuiting ensures that the right expression is not evaluated in this case
Boolean Operator: ! (NOT)

- Flips the state of a Boolean expression
  - !true is false
  - !false is true

Example
- Walk to school if it is not raining
  - if (!isRaining)
    
    // walk to school


Boolean Operator: ! (NOT)

- Flips the sense of a comparison
  - !true is false
  - !false is true

- Generally used with Boolean expressions
The Conditional Operator: `?:`

- Ternary Operator (3 operands)
  - Not a statement!

- Syntax
  - `Boolean_Expression ? Expression_1 : Expression_2`

  - `Expression_1` and `Expression_2` must each evaluate to the same type
    - The result of the `?:` operator evaluates to that type

- Recommend surrounding each expression with parentheses for clarity
Example 1

Equivalent Code Fragments

```java
if (altitude <= 100) {
    System.out.println("Too Low!" );
} else {
    System.out.println("It's ok.");
}
```

```java
System.out.println((altitude <= 100) ? "Too Low!" : "It's ok.");
```
Example 2

Equivalent Code Fragments

- `double pay;
  if (hoursWorked <= 40)
  { pay = hoursWorked * PAY_RATE;
  }
  else
  { pay = 40 * PAY_RATE + (hoursWorked - 40) * 1.5 * PAY_RATE; }

- `double pay = (hoursWorked <= 40) ? (hoursWorked * PAY_RATE) : (40 * PAY_RATE + (hoursWorked - 40) * 1.5 * PAY_RATE);`
Prompt user for chairSize

chairSize > 10
Print "too big"

chairSize < 5
Print "too small"

What is chairSize?

(chairSize >= 5) && (chairSize <= 10)
Print "just right"
More Than Two Branches

```java
if (chairSize < 5) {
    System.out.println("too small");
} else if (chairSize > 10) {
    System.out.println("too large");
} else {
    System.out.println("just right");
}
```
More Than Two Branches

These are equivalent code fragments

```java
if (chairSize < 5)
{
    System.out.println("too small");
}
else if (chairSize > 10)
{
    System.out.println("too large");
}
else
{
    System.out.println("just right");
}
```

```java
if (chairSize < 5)
{
    System.out.println("too small");
}
else
{
    if (chairSize > 10)
    {
        System.out.println("too large");
    }
    else
    {
        System.out.println("just right");
    }
}
```

Only one branch in an if/else if/else if/else will be executed
Nested if statements

```java
if (chairSize < 5)
{
    System.out.println("too small");
}
else if (chairSize > 10)
{
    System.out.println("too large");
    if (chairSize > 100)
    {
        System.out.println("way too large");
    }
}
else
{
    System.out.println("just right");
}
```
If Gotchas

- Always use {} rather than tabs to indicate multi-statement then- or else-blocks

- If you want multiple then-blocks to execute, then don’t use else if, just use a separate if statement
If Gotchas

- Don’t put a semicolon right after the if condition
  
  Wrong:
  - if (x < 5) ;
    action();
  
  action() will always be performed

- Right:
  - if (x < 5) {
      action();
    }

7/6/2012
The switch Statement

Syntax

```java
switch (Expression)
{
    case Case_Literal_1:
        List_Of_Statements_1
        break;

    case Case_Literal_2:
        List_Of_Statements_2
        break;

    // ...

    case Case_Literal_n:
        List_Of_Statements_n
        break;

    default:
        List_Of_Default_Statements
        break;

}
```

- The evaluated type of `Expression` and the type of the `Case_Literals` must be the same
  - Java 6 & 5th Edition Textbook
    - Only `int`, `char`, and `enum` types are supported
  - Java 7 & 6th Edition Textbook
    - Only `int`, `char`, `enum` types, and `String` are supported
    - Strings behave as if using the `equals` method

- The number of cases can be zero or more

- The default case is optional
The switch Statement

Syntax

switch (Expression) {
    case Case_Literal_1:
        List_Of_Statements_1
        break;
        
    case Case_Literal_2i:
    case Case_Literal_2ii:
    case Case_Literal_2iii:
        List_Of_Statements_2
        break;
        
    // ...
        
    case Case_Literal_n:
        List_Of_Statements_n
        break;
        
    default:
        List_Of_Default_Statements
        break;
}

The number of literals assigned to a List_Of_Statements can be one or more

- If any of the literals match the value of the evaluated expression, then that list of statements will be executed
The break Statement

Syntax

switch (Expression) {
    case Case_Literal_1:
        List_Of_Statements_1
        break;
    case Case_Literal_2:
        List_Of_Statements_2
        break;
    // ...
    case Case_Literal_n:
        List_Of_Statements_n
        break;
    default:
        List_Of_Default_Statements
        break;
}
If/Else v. Switch

```java
if (year == 1)
{
    System.out.println("freshman");
}
else if (year == 2)
{
    System.out.println("sophomore");
}
else if (year == 3)
{
    System.out.println("junior");
}
else if (year == 4)
{
    System.out.println("senior");
}
else
{
    System.out.println("unknown");
}

switch(year)
{
    case 1:
        System.out.println("freshman");
        break;
    case 2:
        System.out.println("sophomore");
        break;
    case 3:
        System.out.println("junior");
        break;
    case 4:
        System.out.println("senior");
        break;
    default:
        System.out.println("unknown");
        break;
}
```

7/6/2012
Loops
Loops

- Often you need to repeat an action in a program
Loops Components

- **Loop**
  - Part of a program that repeats

- **Body**
  - The statements being repeated

- **Iteration**
  - A repetition of the body

- **Controlling Condition**
  - A Boolean expression that indicates the loop should repeat

Need More Sandwiches?
- Yes
  - Make Sandwich
  - Distribute Sandwiches
- No
Basic Looping Algorithm

1. Read in value from user
2. While value is valid
   1. Process value into a composite value
   2. Read in a new value from user
3. Output composite value
Loop Termination

- The loop terminates when the controlling expression becomes false
  - Requires the body of the loop to cause this change

- Infinite Loop: a loop with a controlling expression that never becomes false
  - Often caused by a body that does not cause the controlling expression to change to false
Types of Loops

- While Loop
  - General looping, but the body need not be executed at least once

- Do...While Loop
  - General looping, but the body must be executed at least once

- For Loop
  - Looping over a known iteration count

- For Each Loop
  - Looping over a set of values
While Loop

Flowchart

Pseudocode

1. Start
2. While Need More Sandwiches
   1. Make Sandwich
3. Distribute Sandwiches
while Loop

- Syntax
  - \texttt{while (Boolean\_Expression)}
  - \{ \texttt{Body\_Statement} \}

- Execution
  - If the \texttt{Boolean\_Expression} evaluates to true, the \texttt{Body\_Statement} is executed
  - As long as \texttt{Boolean\_Expression} remains true, the \texttt{Body\_Statement} is executed again
  - If the \texttt{Boolean\_Expression} evaluates to false, flow of control continues to the next statement
While Loop Example

Code

```java
System.out.println("Enter a series of positive integers and I will add them.");
System.out.println("Enter a negative number to indicate the end of the series.");

Scanner keyboard = new Scanner(System.in);

int currentValue = keyboard.nextInt();
int sum = 0;

while (currentValue >= 0) {
    sum += currentValue;
    currentValue = keyboard.nextInt();
}

System.out.println("The sum of the positive values is "+ sum);
```

Sample Execution

Enter a series of positive integers and I will add them. Enter a negative number to indicate the end of the series.

```
2
6
2
5
7
2
-1
```

The sum is 24
Sentinel Value

- A special value used to break a loop

- The value cannot match normal values used within the loop, for example
  - Use a negative number to end a loop over positive numbers
  - Use zero to end a loop of non-zero numbers
  - Use a word to end a loop of parsed (converted from string) numbers
While Loop Example

Code

```java
System.out.println("Enter a series of positive integers and I will add them.");
System.out.println("Enter a negative number to indicate the end of the series.");

Scanner keyboard = new Scanner(System.in);

int currentValue = keyboard.nextInt();
int sum = 0;

while (currentValue >= 0)
{
    sum += currentValue;
    currentValue = keyboard.nextInt();
}

System.out.println("The sum of the positive values is " + sum);
```

Sample Execution

Enter a series of positive integers and I will add them. Enter a negative number to indicate the end of the series.

2 6 2 5 7 2
-1
The sum is 24
While v. Do...While Loops

**While Loop**

1. Start
2. Loop Body Statements
3. Controlling Expression
4. If True, go to Loop Body Statements; if False, go to End

**Do...While Loops**

1. Start
2. Loop Body Statements
3. Controlling Expression
4. If True, go to Loop Body Statements; if False, go to End

Diagram:

- While Loop:
  - Start
  - Loop Body Statements
  - Controlling Expression
  - If True, go to Loop Body Statements
  - If False, go to End

- Do...While Loops:
  - Start
  - Loop Body Statements
  - Controlling Expression
  - If True, go to Loop Body Statements
  - If False, go to End
While v. Do...While Loops

**While Loop**

- **Syntax**
  - `while (Boolean_Expression)
    {
      Body_Statement
    }
  
  - Body_Statement may never be executed

**Do...While Loops**

- **Syntax**
  - `do
    {
      Body_Statement
    }
    while (Boolean_Expression);

  - Body_Statement will execute at least once
  - do...while statement always ends with ;
Do...While Loops: The ;

Syntax

- do
  Body Statement
  while (Boolean_Expression);

- do
  {
    Body Statement_1
    Body Statement_2
    // ...
    Body Statement_n
  } while (Boolean_Expression);

Examples

- int i = 1;
  do
    i++;
  while (i <= 10);

- int i = 1;
  do
    {
      System.out.println(i);
      i++;
    } while (i <= 10);

Recommend using block syntax, even for single statement loops
For Loops

- A “syntactic sugar” for a common form of while loop
  - Provides no new features, but is convenient to use

Syntax

- for (Init_Statement; Boolean_Expression; Update_Statement)
  Body_Statement

- for (Init_Statement; Boolean_Expression; Update_Statement)
  {
    Body_Statement_1
    Body_Statement_2
    // ...
    Body_Statement_n
  }
For Loop as While Loop

- For Loop
  ```
  for (Init_Statement; Boolean_Expression; Update_Statement)
  {
    Body_Statement_1
    Body_Statement_2
    // ...
    Body_Statement_n
  }
  ```

- While Loop
  ```
  Init_Statement
  while (Boolean_Expression)
  {
    Body_Statement_1
    Body_Statement_2
    // ...
    Body_Statement_n
    Update_Statement
  }
  ```

- Like the while loop, if the Boolean_Expression starts out false, the body will not be executed
For Loop as While Loop

- These three loops are equivalent (except for the scope of \( i \))
  - For Loop
    - ```java
      int i;
      for (i = 1; i <= 10; i++)
      {
        System.out.println(i);
      }
    ```
  - ```java
    for (int i = 1; i <= 10; i++)
    {
      System.out.println(i);
    }
    ```
  - While Loop
    - ```java
      int i = 1;
      while (i <= 10)
      {
        System.out.println(i);
        i++;
      }
    ```
Can declare the loop variable inside the `Init_Statement` of the `for` statement:

```java
for (int i = 1; i <= 10; i++)
{
    System.out.println(i);
}
```
For Each Loop

- Syntax
  
  ```java
  for (Type variable : Set)
  {
    Body_Statement
  }
  ```

- Arrays can be used as Set
  
  ```java
  double[] readings = new double[50];
  // ...
  for (double reading : readings)
  {
    System.out.println(reading);
  }
  ```

- Note that only the value of each element is accessible in this loop
  - Indices are not available
Break Statement

- Returns control to the statements that follow the current loop
  - Aborts the loop

- Syntax:
  - break;
Continue Statement

- Advances to the next iteration of the current loop
  - Skips the rest of the statements in this iteration
  - Still performs the Update_Statement in for loops
  - Still tests the controlling Boolean_Expression in all loops

- Syntax:
  - continue;
The Very Large Array

Arrays to the Rescue

- An array is a collection of items of the same type
  - Can store any type (primitives, classes, arrays)

- They are like a list of variables, with a compact naming scheme
  - Each item in the array is an element

- Arrays are special kinds of objects in Java
  - Like a class type, not a primitive type
Declaring an Array

- **Syntax**
  - `Type[] arrayVar = new Type[Integer_Expression];`

- `Type` indicates the kinds of values the array can hold
  - Can be primitive, class, or other array types

- The value of `Integer_Expression` determines the number of elements the array can hold
Declaring an Array

- Can also assign a list of values directly to an array
  - `int[] scores = { 47, 52, 94 };`

- This is equivalent to
  - `int[] scores = new int[3];`
    - `scores[0] = 47;`
    - `scores[1] = 52;`
    - `scores[2] = 94;`
Indexing

- Syntax
  - `arrayVar[Integer_Expression]`

- These variables are known as indexed variables, subscript variables, array elements, or elements

- The value of `Integer_Expression` determines which element is being accessed
Indexing

- Integer Expression can be any integer expression
  - An integer: values[3]
  - An integer variable: values[index]
  - An expression that evaluates to int: values[index * 3]

- Can use array elements like any other variable
  - values[3] = 68;
  - values[4] += 3;
  - System.out.println(values[16]);
Indexing

- Where have we seen indexing before?
  - Strings, e.g. `myStr.charAt(5)`

- Like Strings, array indices always start at zero

- Strings are read-only, while arrays are read/write
Indexing

- Like Strings, arrays have a size (number of elements)
  - String: `myStr.length()` // method call, uses ()
  - Array: `myArray.length` // instance variable, no ()

- You cannot change the length of an array
  - `myArray.length` is read-only

- The value of `length` is equal to the number of elements in the array
  - `int[] values = new int[20]` // `values.length` is now 20
Array Gotchas: Assignment

- Need to assign an array to a variable before giving its elements values
  - `int[] scores;`
  - `scores[2] = 5; // Compiler Error`

- The code will not compile
Array Gotchas: Assignment

- Assigning existing arrays to other variables does not create new copies
  - The second variable will refer to the same instance as the original variable

- Consider this example:
  - ```java
  int[] myArr1 = new int[5];
  myArr1[2] = 5;
  int[] myArr2 = myArr1;
  myArr2[2] = 3;
  System.out.println(myArr1[2]); // Prints 3
  ```

- Why?
  - Arrays are reference types
Array Gotchas: Bounds

- Need to stay within the bounds of the array
  - `int[] scores = new int[7];
scores[7] = 5; // Runtime Error`

- The code will compile, but will produce an Array Index Out of Bounds error at runtime
Array Gotchas: Equality

- Like Strings (and any other class typed object), don’t use `==` to test equality
  - It will not behave like `==` does for primitives

- Need to write your own code if array equality testing is required
Sorting

- Given an array of numbers, sort the numbers into ascending order

- For example

| 4 | 7 | 3 | 9 | 6 | 2 | 8 |

becomes

| 2 | 3 | 4 | 6 | 7 | 8 | 9 |
Selection Sort

- **Pseudocode**
  - `for` (index = 0; index < length; index++)
    ```
    for (index = 0; index < length; index++)
    {
        Inner Loop
        {
            // 1. Find index of smallest value of array
            //    between index and end of array
            // 2. Swap values of current index and the
            //    index with the smallest value
        }
    }
    ```

- Java code is available on the class website (SelectionSort.java)
Bubble Sort

- **Pseudocode**
  - while the array is not sorted
    - for (index = 0; index < a.length; index++)
      - if (a[index] > a[index + 1])
        - interchange values of a[index] and a[index + 1]

- Java code is your solution for Lab 5
Sorting

- Other sorting algorithms exist
  - Bubble Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort

- Each uses different algorithms and has different performance characteristics
Multidimensional Arrays

- Arrays having more than one index tend to be useful
  - Tables
  - Grids
  - Matrices
  - Images
n-D Arrays

- **Declaration**
  - \( Type[]...[] \) arrayVar1, arrayVar2, …;

- **Array Creation**
  - arrayVar = new Type[Int.Expr1]...[Int.ExprN];

- **Element Assignment**
  - arrayVar[Int.Expr1]...[Int.ExprN] = Expression;
n-D Arrays & Loops

- Use a single loop on 1-D arrays
  ```java
  int[] scores = { 4, 6, 12, 2, 6 ];
  for(int i = 0; i < scores.length; i++)
  {
    System.out.println(scores[i]);
  }
  ```

- Use nested loops on 2-D arrays
  ```java
  int[][] table = new int[4][5];
  for (int row = 0; row < table.length; row++)
  {
    for (int column = 0; column < table[row].length; column++)
    {
      System.out.println(table[row][column]);
    }
  }
  ```
2-D Array Lengths

- `int[][][] table = new int[4][5];`
  - `table.length` is the number of rows: 4
  - `table[row].length` is the number of columns: 5
Arrays Summary

- Arrays are class-like variables
  - Contain a list of values of all the same type
  - These values are indexed using integer expressions within [ ]’s: myArr[index]
    - Indices Range: 0 <= index < myArr.length

- Can iterate through the values of an array using a loop
  - Use nested loops for multidimensional arrays

- Sorting is the process of organizing the values of an array into ascending order
Debugging Techniques

Photo # NH 96566-KN  First Computer "Bug", 1945

9/9

0800  Antennas started
1000  Stop - antennas

13:52  (03) MP - ME
       (03) PRO - 2.130976415

Relays 6-2 in 033 failed special speed test

In test:

Relays changed

1100  Started Cosine Tape (Sine check)
1525  Started Multiplier Adder Test

1545  Relay #70 Panel F

First actual cause of bug being found.

1700  Check done.
Testing & Debugging

- Sometimes designing and implementing that design is not enough
  - Test the program to verify it behaves correctly
  - Debug the program to resolve any problems
Types of Bugs

- Syntax Error: A violation of the programming language’s grammar
  - In this case, the program will not compile or run

- Examples:
  - `int a = 5,`
  - `int b = 0.5;`
  - `c +`
Types of Bugs

- Run-Time Error: an error that is detected during program execution
  - The program will compile, run to the point of failure, and then crash

Example:

- int n1 = 5;
  int n2 = 0;
  int n3 = 2 + n1 / n2;
Types of Bugs

- Logic Error: a mistake in the program caused by the underlying algorithm
  - The program will compile and run, but behave unexpectedly

Examples:

- `double unitCost = 2.50;
  int quantity = 5;
  double totalCost = unitCost / quantity;

- `double a = 1.0, b = 2.0, c = 3.0;
  double x1 =
    (-b + Math.sqrt(b * b - 4 * a * c)) / 2.0`
Loop Bugs

- Two most common mistakes in loops
  - Off-by-One Error: error involving the discrete equivalent of a boundary condition
    - Typically a logic error but can be a runtime error
  - Infinite Loop: a loop that never terminates
Resolving Off-by-One Errors

- If you want to execute for \( n \) times
  - Use a for loop
  - If you want to start at 0
    - Use a strictly less than condition: \(< n\)
  - If you want to start at 1
    - Use a less than or equals condition: \(\leq n\)

- When testing, be aware of the this error possibility
Resolving Infinite Loops

- Remember to include some condition that will terminate the loop
  - Sometimes this is best as the controlling Boolean_Expression
  - Sometimes this is best using a break statement
Documentation & Style

- Meaningful Names
- Indenting
- Comments
- Named Constants
Meaningful Names

- Assigning names that are easy to understand reduces the amount of other documentation required.
Indenting

- Block: A grouping construct of code, often bounded by { and }

- Indenting Style Rules
  - Indent by one level within a block
    - 
      ```java
      { 
      System.out.println("Hello!");
      }
      ```
  - Indent by one level a line continuation
    - String aBunchofStuff = "This is a bunch" + "of text and a number: " + n1 + ". Isn't it?";

- Compiler ignores indenting, use { }’s to indicate a block
Comments

- Provides documentation
- Compiler ignores them

Examples:
- // This is a single line comment
- // This is a pair of // single line comments
- /* This is a * multi-line comment */
Named Constants

- Syntax
  - public static final Type VARIABLE_NAME = value;

- Placement: Inside the class, outside the main method
- Style: All letters capitalized, words underscore-separated

- Example
  - public class SomeProgram
    {
      public static final int DAYS_PER_WEEK = 7;
      public static void main(String[] args)
      {
        // ...
    }
Leftovers
Operators

- **Unary v. Binary v. Ternary**
  - **Unary**: 1 operand
    - E.g. `myVar++`
  - **Binary**: 2 operands
    - E.g. `score1 + score2`
  - **Ternary**: 3 operands
    - E.g. `(grade > 5) ? “Decent” : “Poor”`
Order of Operations

- Parentheses are evaluated first, then from highest to lowest precedence:
  - Unary: +, -, ++, --, !
  - Unary: new, (Type)
  - Binary: *, /, \\
  - Binary: +, -
  - Binary: <, >, <=, >=
  - Binary: ==, !=
  - Binary: &
  - Binary: ^
  - Binary: |
  - Binary: &&
  - Binary: ||
  - Ternary: ?: 
  - Binary: All assignment operators (=, +=, ...)
Questions?
Logistics

- Monday:
  - The Midterm

- I will be grading your turned-in assignments over the weekend
  - Priority given to Labs