Lab Observations

• Saw lots of good pairing
  – Work on one program together
• Some partners didn’t show up – need to attend lab together
• Lots of questions and uncertainty
  – Don’t be afraid to ask questions in class
  – If you don’t understand something, you’re probably not the only one

Pair Partners

IMPORTANT

If you have lost contact with your partner, or you don’t have a partner, see me after class.

// SimpleInput.java
// read numbers from the keyboard
import tio.*; // use the package tio

class SimpleInput {
  public static void main (String[] args) {
    int width, height, area;
    System.out.println("type two integers for" + " the width and height of a box");
    width = Console.in.readInt();
    height = Console.in.readInt();
    area = width * height;
    System.out.print("The area is ");
    System.out.println(area);
  }
}
Calling predefined methods

- A **method** is a group of instructions with a name
  - We’ve seen `main()`, `System.out.println()`
- We **execute a method by calling it**
  - We call a method by putting its name in the program where we want it to be executed
- **Method names don’t have to be unique**
  - Identified by the object name - `System.out.println()`
- **Function** is another name for method

Calling predefined methods

- Many methods take inputs: **parameters**
- Parameters are **passed** to the method by placing them between the parentheses
- Example: `System.out.println("Hello");`
  - "Hello" is the parameter passed to `System.out.println()`
- Multiple parameters are separated by commas
  - `someFunction(a, b);`
- Some methods also return a value.
  - `x = Math.sqrt(y);`

**print() and println()**

- `System.out.print()` and `System.out.println()` print out strings and the primitive types
  - What’s a primitive type?
- Difference: `println()` puts a newline at the end
- Explicit newline is represented by ‘\n’, as in `System.out.print("Hi\nBrian\n");`
  - Common mistake - '/n' instead of 'n'
print() and println()

- Concatenation with ‘+’
  - ‘+’ allows multiple things in a print( ) statement
  - System.out.print("The value is: "+value);
  - Common mistake: ‘,’ instead of ‘+’
- Be careful with numeric types
  - Given int a = 5, b = 7;
  - System.out.println("The value is: "+a+b); prints out “The value is: 57”
  - System.out.println("The value is: "+(a+b)); prints out “The value is: 12”
  - System.out.println(a+b); prints out “12”

System.out.println("type two integers for the width and height.");
System.out.print("type two integers for the");
System.out.println(" width and height.");

Illegal
System.out.println("type two integers for the width and height.");

// SimpleInput.java
// read numbers from the keyboard
import tio.*;    // use the package tio
class SimpleInput {
    public static void main (String[] args) {
        int width, height, area;
        System.out.println("type two integers for" +
        " the width and height of a box");
        width = Console.in.readInt();
        height = Console.in.readInt();
        area = width * height;
        System.out.print("The area is ");
        System.out.println(area);
    }
}
Number Types

- Two basic representations for numbers
  - Integer: whole numbers
  - Floating point: fractional numbers and very big numbers
- Bit
  - The smallest element of storage in a computer
  - Can be either 0 or 1
  - Bigger numbers are stored as a sequence of bits

Representing Numbers with Bits

- A sequence of bits is interpreted as a binary number
  - 00, 01, 10, 11 binary = 0,1,2,3 in decimal
  - Read Appendix A
- A byte is 8 bits
  - Smallest addressable unit in a computer
  - Can contain any number between –128 and 127
  - Why?

Integer Types

<table>
<thead>
<tr>
<th>Type</th>
<th>#bits</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>-128 to +127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>0 to 65536</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2147483648 to +2147483647</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>-9223372036854775808 to +9223372036854775807</td>
</tr>
</tbody>
</table>
### Floating Point Types

<table>
<thead>
<tr>
<th>Type</th>
<th>#bits</th>
<th>Approximate Range of Values</th>
<th>Approximate Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32</td>
<td>+/- (10^{-45}) to +/- (10^{+45})</td>
<td>7 decimal digits</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>+/- (10^{-324}) to +/- (10^{+308})</td>
<td>15 decimal digits</td>
</tr>
</tbody>
</table>

Java displays floating point numbers in two ways:
- With a decimal point: 34.576
- Similar to scientific notation:
  - \(2.479e23\) represents \(2.479 \times 10^{23}\)
  - \(3.75e-15\) represents \(3.75 \times 10^{-15}\)

---

### Char

- A character is a special type
- Holds integer values that represent Unicode characters
  - English uses only a very small subset
  - Unicode developed to support all languages and character sets

### Some Examples

<table>
<thead>
<tr>
<th>Char</th>
<th>ASCII Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>97</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
</tr>
<tr>
<td>'A'</td>
<td>65</td>
</tr>
<tr>
<td>'B'</td>
<td>66</td>
</tr>
<tr>
<td>'C'</td>
<td>67</td>
</tr>
<tr>
<td>'0'</td>
<td>48</td>
</tr>
<tr>
<td>'1'</td>
<td>49</td>
</tr>
<tr>
<td>'9'</td>
<td>57</td>
</tr>
<tr>
<td>'&amp;'</td>
<td>38</td>
</tr>
<tr>
<td>'+'</td>
<td>43</td>
</tr>
</tbody>
</table>

### Some Important Properties
- Alphabetic characters (A-Z, a-z) are in order
- Numeric characters (0-9) are in order

### Some Special Characters:
- \(\backslash n\) new line
- \(\backslash t\) tab
- \(\backslash '\) single quote
- \(\backslash "\) double quote
- \(\backslash \) backslash
Example: Special chars

The statement:

```java
System.out.println("Brian says, \\
Hello, class");
```

Prints:

```
Brian says, “Hello, class”
```

---

Numeric Literals

- Integer literals
  - Default to type int
  - Can be specified as long by ending with ‘L’
  - 24, 1003, 123887698888L
- Floating point literals
  - Default to type double
  - Can be specified as float with ‘F’
  - 3.7, 2.9, 3.1416F, 1358494.34792098

---

Numbers vs. Strings

- These are all different
  - The number 50
  - The string “50”
  - The number 50.0
  - The character with value 50 (happens to be ‘2’)
- The computer stores them using different bit patterns
- Different patterns are better for different types of data manipulation
- Java types tell the computer how to interpret the bit patterns
Arithmetic Expressions

- Operators:
  - Addition: +
  - Subtraction: -
  - Multiplication: *
  - Division: /
  - Modulus (remainder): %

- Types: char, byte, short, int, long, float, double

Modulus (Remainder)

- There is an error in the book on page 37
  - Says that % on floating point works differently – this is incorrect
  - Actually % calculates remainder on integer and floating point:
    - 5 % 3 = 2
    - 5.0 % 3.0 = 2.0
    - 5.1 % 3.0 = 1.2

Rules of Mixed-mode Arithmetic

1. An arithmetic operation on objects of the same type yields a result of that type
2. An arithmetic operation on objects of different types first promotes smaller types to larger type
   - Any operand is a double --> promoted to double
   - Otherwise, any float --> promoted to float
   - Otherwise, any long --> promoted to long
   - Otherwise, any int --> promoted to int
   - And then rule 1 applies
Rules of Mixed-mode Arithmetic

Given these declarations:

```
Byte aByte = 1;
int anInt = 2;
float aFloat = 1.5;
double aDouble = 2.4;
```

Then in the expressions:

```
aByte + anInt  --> promoted to int
aByte + aFloat --> promoted to float
aFloat + aDouble + anInt --> promoted to double
```

Details

- Any result value that is too big for the result type will be undefined
  - Solution: force promotion when necessary by using a variable or literal of the larger type or by casting one operand to a suitably larger type
  - Example: (float)5, (long)a, 5.0, 4f
- Integer types storing floating point values will have the fractional part truncated
  - Towards 0
  - 4 / 5 = 0 but 4.0/5 = 0.8

```java
// MakeChange.java - change in dimes and pennies
import tio.*; // use the package tio
class MakeChange {
    public static void main (String[] args) {
        int price, change, dimes, pennies;
        System.out.println("type price (0 to 100): ");
        price = Console.in.readInt();
        change = 100 - price; // how much change
        dimes = change / 10; // number of dimes
        pennies = change % 10; // number of pennies
        System.out.println("The change is ");
        System.out.println(dimes + " dimes, ");
        System.out.println(pennies + " pennies");
    }
}
```
Type Conversion

- Implicit (in mixed-mode arithmetic)
- Explicit (casting)
- Widening
  - From “smaller” to “larger type”
  - All information is retained
- Narrowing
  - From “larger” to “smaller” type
  - Information may be lost
  - Result may be meaningless
- Common mistake: int z = 3.0/4.0;

Assignment Operator

- `<variable> = <rightHandSide>;`
- What happens?
  - Right hand side is evaluated
  - The result is placed in the variable `<variable>`
- Examples:
  - a = 0;
  - a = b + c;
  - a = Console.in.readInt();
  - a = b = c;

More Assignment Operators

- `+= , -= , *= , /= , %= , >>= , <<= , &= , ^= , |=`
- `+=` is pronounced “plus equals”, etc.
- All but `=` are shorthand
- Example
  - `a += b;` is shorthand for `a = a + b;`
  - `a += c * d;` is shorthand for `a = a + (c * d);`
  - `a *= b + c;` is shorthand for `a = a * (b + c);`
- The others work the same way
Increment and Decrement Operators

- ++ is shorthand for “add one to the variable”
  - i++; and ++i; are shorthand for i = i + 1;
- -- is shorthand for “subtract one from the variable”
  - i--; and --i; are shorthand for i = i - 1;
- Location determines order of evaluation
  - int a, b = 0;
  - a = ++b; // result: a = 1 and b = 1;
  - a = b++; // result: a = 0 and b = 1;

Order of Evaluation

- In expressions with multiple operators, order matters!
- Example:
  - j = (3 * 4) + 5; // result: j = 17
  - j = 3 * (4 + 5); // result: j = 27

Precedence and Associativity

- Precedence specifies which operators are evaluated first
- Associativity specifies order when operators have equal precedence
- Parentheses ( ) override these
  - They force whatever is inside to be evaluated as a unit
- *, /, and % have equal precedence
- + and - have equal precedence
- *, /, and % have higher precedence than + and -
- Examples:
  - x = 3 * 4 * 5;
  - x = (3 + 4) * 5;
  - x = 40 / (5 * 2);
- Look at table on page 45 and Appendix B for details
Precedence and Associativity

• Expressions can get rather complicated
  \[ a + \frac{b}{c} \times d - \frac{7}{3}; \]
• Use parentheses if you are unsure, or to make your intent obvious
  \[ (a + \frac{(b/c) \times d)}{(7/3)}; \]

Programming Style

• Comments
  – At the top of every file
  – At the top of every class definition
  – At the top of every method definition
  – At the top of every non-trivial block of instructions
• Identifiers should be meaningful
• Readability, understandability, clarity, elegance

Java Naming Conventions

• Class names start with uppercase and embedded words are capitalized:
  – HelloWorld, TypeConvert
• Methods and variables start with lowercase and embedded words are capitalized
  – readInt, data, toString, loopIndex
• $ should not be used and _ marks you as an old C programmer
<table>
<thead>
<tr>
<th>Software Lifecycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem Analysis and Specification</strong></td>
</tr>
<tr>
<td>- What needs to be done?</td>
</tr>
<tr>
<td><strong>Design</strong></td>
</tr>
<tr>
<td>- How should it be done</td>
</tr>
<tr>
<td>- Creation of a solution (algorithm)</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
</tr>
<tr>
<td>- Turn the algorithm into a program</td>
</tr>
<tr>
<td><strong>Verification</strong></td>
</tr>
<tr>
<td>- Does the program do what it is supposed to do?</td>
</tr>
<tr>
<td>- Does it not do what it is not supposed to do?</td>
</tr>
<tr>
<td><strong>Maintenance</strong></td>
</tr>
<tr>
<td>- Change what the program does</td>
</tr>
<tr>
<td>- Includes both bug fixes and modifications</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Analysis and Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What it does:</strong></td>
</tr>
<tr>
<td>- Clearly defines problem - what is/is not being solved</td>
</tr>
<tr>
<td>- Refines imprecise problem to one that is solvable given existing constraints</td>
</tr>
<tr>
<td>- Constitutes an agreement on what is to be done</td>
</tr>
<tr>
<td>- May discover problems</td>
</tr>
<tr>
<td>- Inconsistency, vagueness, impossibility</td>
</tr>
<tr>
<td>- Leads the way to the solution</td>
</tr>
<tr>
<td>- May contain desirable and optional items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Analysis and Specification (cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What it does (cont.):</strong></td>
</tr>
<tr>
<td>- Should be specific enough to be testable, so you know if/when the problem has been solved</td>
</tr>
<tr>
<td>- Often done inadequately</td>
</tr>
<tr>
<td><strong>What it doesn’t do:</strong></td>
</tr>
<tr>
<td>- Specify how to solve the problem</td>
</tr>
<tr>
<td><strong>Important parts of a problem specification</strong></td>
</tr>
<tr>
<td>- A list of inputs</td>
</tr>
<tr>
<td>- A list of constants</td>
</tr>
<tr>
<td>- A list of outputs</td>
</tr>
</tbody>
</table>
Problem Analysis and Specification (cont.)

- Examples:
  - Yes: Should calculate change in dime/penny land
  - No: Should be fast
  - Yes: Should run in less than 10 seconds
  - No: Should use quicksort
- For the programming assignments, I will provide a problem specification
  - This specification may be incomplete
  - It is your job to analyze and understand the specification and refine it as necessary

Design

- What it does:
  - Clearly specifies how the problem will be solved
  - Allows developers to determine what resources will be needed to solve the problem
  - Hopefully solves all problems that could arise in the development of the software component
  - Is used as a recipe for doing the actual coding
- What it doesn’t do:
  - A design is not code and does not contain any code.
    - May contain pseudocode
  - A design is not specific to any language, although it usually is specific to a type of language

Design (cont.)

A software design typically has 3 parts:
1) Identification of the data objects that are required to solve the problem
2) Identification of the operations that must be applied to the data objects in order to solve the problem
3) Construction of a detailed sequence of steps (an algorithm) that specifies how the operations can be applied to the data objects to solve the problem
Implementation

• Once the design is complete, coding can begin
  – Given a good design, this should be very straightforward
  – All hard problems should have been worked out in the design stage
  – New hard problems should send the project (temporarily) back to the design stage

Implementation (cont.)

• Good code should be
  – Correct
  – Readable and Understandable
  – Modifiable
  – Ideally: Reusable

Implementation (cont)

• Good programs should be:
  – Well structured
    • Break programs into meaningful parts
    • Strive for simplicity and clarity
  – Well documented (commented)
    • Good comments before each program and/or function
    • Good comments before each important part of a program/function
    • Use meaningful identifiers (function and variable names)
  – Aesthetically pleasing
    • Space things out and use blank lines between logical blocks
    • Use alignment and indentation to emphasize relationships
Verification

- Each program and subprogram should be tested against its requirements
  - To see that it does what it is supposed to do
  - To make sure that it does not do what it is not supposed to do
- Tests should include correct and incorrect inputs
  - Even nonsense inputs
- Regression tests
  - Make sure that new changes don’t break old functionality

Maintenance

- Bugs are found that need to be fixed
- Requirements change
- Components are reused
- Enhancements are made
- Generally accomplished by repeating the first four steps
- Most software development effort is maintenance

Example: Problem

Specification/Analysis

**Problem:** Write a program that, given diameter of a circle, computes the area and circumference

**Description:** Compute and output the area and circumference of a circle given the diameter.

**Input:** Diameter of the circle

**Outputs:** Area and Circumference of the circle

**Constants:** Pi, and maybe formulas for area and circumference of circles
Example: Design

- **Data objects:**
  - **Variables:**
    - Real: diameter, circumference, area, radius
  - **Constants:**
    - Real: pi
  - **Operations:**
    - $\text{radius} = \text{diameter} / 2$
    - $\text{circumference} = 2 \times \pi \times \text{radius}$
    - $\text{area} = \pi \times \text{radius}^2$

Algorithm:
1. Get diameter from user
2. Calculate radius
3. Calculate circumference
4. Calculate area
5. Print out values of circumference and area

Example: Implementation

```java
// Calculate area and circumference
import tio.*; // use the package tio

class CalAreaCircumference {
    public static void main(String[] args) {
        int diameter;
        double radius, circumference, area, pi;
        pi = 3.14159625;

        // Get diameter from user
        System.out.println("Diameter: ");
        diameter = Console.in.readInt();
```
Example: Implementation (cont)

    // Calculate
    radius = diameter / 2.0;
    circumference = 2 * pi * radius;
    area = pi * radius * radius;

    // Print results
    System.out.println("Circumference = ");
    System.out.println(circumference);
    System.out.println("Radius = ");
    System.out.println(radius);
}

Example: Verification

• Run the above program with:
  – No input (shouldn't be allowed)
  – Negative input (shouldn't be allowed)
  – Positive input (should work)
  – Fractional input (should work)
  – Very large input (should work)
  – What else?