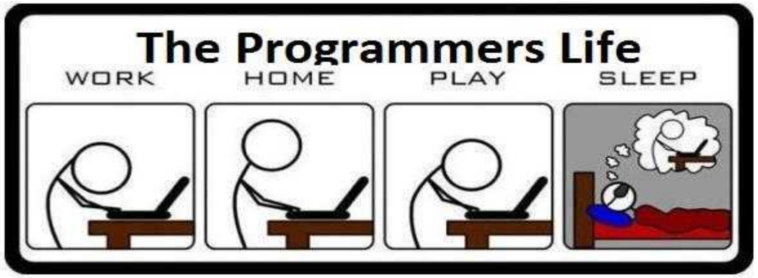
Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Elementary Programming



CS2011: Introduction to Programming I

Designing and Writing Programs

Program Design

- Writing a program has two basic steps:
  - Design a strategy for solving the problem
    - designing an algorithm
    - using pseudocode
  - Using a programming language to implement the algorithm.
    - writing and testing the actual program in Java

Algorithms

- ***algorithm***: a sequence of steps that lists the actions involved in solving a problem.
  - the sequence of steps must be:
    - ***unambiguous*** (the directions are precisely clear at each step, there is no guessing as to how the problem will be solved.)
    - ***executable*** (the instructions are something that the computer can actually carry out)
    - ***terminating*** (the sequence of steps will eventually come to an end)

- an algorithm is a sequence of unambiguous, executable, and terminating steps, that describe how a problem is to be solved.

Pseudocode

- Algorithms can be described in “natural languages” listing the steps and formulas involved.
  - 1. Read the circle's radius
  - 2. Compute the area using the following formula:

    ```
    area = radius x radius x PI
    ```
  - 3. Display the result

- Algorithms can also be described using pseudocode (natural language mixed with some programming code)
  - 1. Read the radius

    ```
    radius = get user input
    ```
  - 2. Compute the area

    ```
    area = radius * radius * PI
    ```
  - 3. Display the results

    ```
    print area
    ```

Designing Basic Programs

Basic Program Design

- Given the previous algorithm we can write a Java class called `ComputeArea` whose outline is as follows:

```
public class ComputeArea {
    //Code given later
}
```

Basic Program Design

- Every program must have a main method:

```
public class ComputeArea {
   public static void main(String[] args) {
      //Step 1: Read the radius

      //Step 2: Compute the area

      //Step 3: Display the area
   }
}
```

Basic Program Design

- The last steps are to get a value for the radius, compute the area, display the results.
- We don't know yet how to read data from the console, so we will just assign a default value of 20 to the radius
- We compute the area by using the formula given previously
- Then display the results using the System.out.println() statement.

Basic Program Design

```
public class ComputeArea {
   public static void main(String[] args) {
      double radius; // Declare radius
      double area; // Declare area

      // Assign a radius
      radius = 20; // radius is now 20

      // Compute area
      area = radius * radius * 3.14159;

      // Display results
      System.out.println("Area of a circle with radius " +
         radius + " is " + area);
   }
}
```

Variables

Variables

- Programs store data in variables..
  - ***variable***: a name that represents a value stored in the computer's memory (RAM)
- Variables should use **descriptive names**:
  - i.e. use radius and area using names like x and y.
- Variables must be declared before they can be used:
  - a variable declaration states the type of the variable and the name.
  - the compiler will allocate memory based on the size required by the type.
  - ex: `int x`, `double area`

Declaring Variables

- Variable Declaration Syntax:

```
datatype variableName;
```

- Variable Declaration Examples:

```
int count;
double radius;
```

- Variables with the same data type can be declared together separated by commas
  - syntax: `datatype variable1, variable2, ..., variablen;`
  - example: `int i, j, k;`

Type of Variables

- A variable's ***data type*** determines what kind of data (and ranges of data) the variable can hold.
  - choose the best type for your data.

- ***primitive data types:*** the most basic data types available.
  - `byte, short, int, long, float, double, boolean, char`
  - primitive types begin with a lowercase letter (e.g. int).

- A ***class (or reference) type*** is used for a class of objects and has both data and methods.
  - `String` is a class / reference type.
  - Class types always begin with an uppercase letter.

Initializing Variables

- Variables also need to be initialized (given a value) before they can be used.

- Declare and initialize in one step:

```
int count = 1;
```

  - this is the preferred way in the value is known ahead of time.

- Declared and initialize separately:

```
int count;
count = 1;
```

- Variables of the same type can be declared and initialized using a shorthand form:

```
int i = 1, j = 2;
```

More Variable Facts

- Variable values can be changed at any time.
  - Example:

```
int x = 10;
x = 7;
x = 5;
```

  - NOTE: Don't declare the data type again since this will cause a compile error.

- The data type cannot be changed once it has been assigned.
  - Example:

```
int x = 10;
double x = 56.4;
```

← Causes a compile error.

- ***scope:***
  - The part of the program where the variable can be referenced.
  - Starts from where the variable is declared and ends at the end of the block that contains the variable.

Printing Variables

- Use `System.out.println` or `System.out.print`
- Example:
  - `System.out.println("The amount is: " + amount);`

Assignment Statements and Expressions

Assignment Statements and Expressions

- ***assignment statements (assignment expressions)*** assign a value to a variable.

- ***assignment operator***: `=`

- Assignment Statement Syntax:

  ```
  variable = expression
  ```

  - An ***expression*** is a computation involving values, variables, constants, and operators that evaluate to a single value.

Assignment Statements and Expressions

- Examples
  - `int y = 1;`
  - `double radius = 1.0;`
  - `int x = 5 * 2 * 2;`
  - `x = y + 1;`
  - `double area = radius * radius * 3.14159`

- NOTE: Assignment is always RIGHT TO LEFT.

Assignment Statements and Assignment Expressions

- Variables can appear on the LHS (left-hand side) or the RHS (right-hand side) of the assignment operator (=).
  - LHS variables are being assigned a value from the RHS.
  - RHS variables are being used in the RHS expression, or are being assigned to another variable on the LHS.
  - Example: `x = y`; assigns the value of y to the variable x.

- Variables can be used in expressions.
  - `x = 2 + (4 * y) – (4 / z);`

- The same variable can be used on both sides of the assignment in the same statement.
  - What is the value of x after the second statement?

```
int x = 1;
x = x + 1;
```

- The variable name MUST be on the left of the assignment operator:

  `2 = x;` //would be wrong

Assignment Statements and Expressions

- The same value can be assigned to multiple variables using the following shorthand:

```
i = j = k = 2;
```

- The previous is equivalent to:

```
k = 2;
j = k;
i = j;
```

Named Constants

Name Constants

- A ***constant*** represents permanent data that never changes.

- Constants differ from variables, because the value of a variable can change throughout a program, but the value of a constant can never change.

- Constants MUST be declared and initialized in the same statement

- Constant Declaration Syntax:

```
final datatype CONSTANT_NAME = value;
```

- Example:

```
final double PI = 3.14159
```

Named Constants

- Benefits of using Constants:
  - Constants declared at the top of the code are easily identified.
  - Repeated values do not have to be typed over and over.
  - Changing the value of the constant will update all subsequent uses of the constant throughout the code.
  - Descriptive names for constants make the program easier to read.

Identifiers and Naming Conventions

Identifiers

- ***identifier***: an entity in a program which can be given a custom name by the programmer
  - classes, variables, constants, methods, packages

- Always choose meaningful, descriptive names.
  - helps maintain code comprehension, maintainability and readablility.

- Identifier requirements:
  - Identifiers can only have letters, digits, and underscores.
  - Must start with a letter or underscore.
  - Cannot be a keyword.
  - Can be any length.

Naming Conventions: Variables and Methods

- Variable and method names should follow these conventions:
  - First letter should be a lowercase letter.
  - Multiple words should be concatenated into a single word, capitalize the first letter of every word except the first.

- Examples:
  - `areaOfCircle`
  - `addTwoNumbers`
  - `showMessageDialog`

Naming Conventions: Class Names

- Classes should follow these conventions:
  - Capitalize the first letter.
  - Multiple words should be concatenated together and the first letter of every word should be capitalized.

- Examples:

```
ComputeArea
ComputeAreaWithConstant
```

Naming Conventions: Constants

- Constants should use the following conventions:
  - Capitalize every letter.
  - Multiple words should be concatenated together and separated by underscores. All letters should be capitalized.

- Examples:

```
PI
RADIUS_OF_EARTH
```

Reading User Input

`import` Statements

- Java has a lot of utilities built into the language.

- Some of the most common classes / utilities are included automatically in every program you write:
  - `System`, `Math`, etc.

- Most have to be imported explicitly:
  - `Scanner`, `Random`, and many others.

`import` Statements

- Import statements:
  - bring an external class into the scope of your program.
  - make available all public items in the external class.
  - appear at the very top of your program.
  - can be used to import Java libraries and libraries written by other programmers.

Console Input with the `Scanner` Class

- The `Scanner` class is used to read ***user input*** from the command line console.

- Import the `Scanner` class: `import java.util.Scanner`

- Create a Scanner object:
  - `Scanner input = new Scanner(System.in);`
  - **`input` here is a variable name for the **`Scanner`** object, and can be anything you want as long as it follows the identifier naming rules.

- Scanner has built-in **methods** to read data of various types. This week we will only worry about reading ***integers*** and ***floating-point*** values:
  - `nextInt()` reads integers from the console.
  - `nextDouble()` reads floating point values from the console.

Console Input with the `Scanner` Class

- Now we can update the previous example by allowing the user to enter the radius.

```
import java.util.Scanner; // Scanner in the java.util package

public class ComputeAreaWithConsoleInput {
   public static void main(String[] args) {
      // Create a Scanner object
      Scanner in = new Scanner(System.in);

      // Prompt the user to enter a radius
      System.out.print("Enter a number for radius: ");
      double radius = in.nextDouble();

      // Compute area
      double area = radius * radius * 3.14159;
      // Display results

      System.out.println("Area of a circle with radius " +
            radius + " is " + area);
   }
}
```

Redundant Input Objects

- Avoid creating multiple instances of the Scanner class. You only need one to handle all user input.

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);
System.out.print("Enter a double value: ");
double v2 = input1.nextDouble();
```

BAD CODE

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();
System.out.print("Enter a double value: ");
double v2 = input.nextDouble();
```

GOOD CODE

Numeric Data Types and Operations

Numeric Types

Name	*Range*	*Storage Size*
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: $-3.4028235E + 38$ to $-1.4E - 45$ Positive range: $1.4E - 45$ to $3.4028235E + 38$	32-bit IEEE 754
`double`	Negative range: $-1.7976931348623157E + 308$ to $-4.9E - 324$ Positive range: $4.9E - 324$ to $1.7976931348623157E + 308$	64-bit IEEE 754

Integer Overflow / Underflow

- If a variable or constant is assigned a value that is too large (or too small) for its data type overflow will occur.

- Example 1:

```
int value = 2147483647 + 1;
//The value will actually be -2147483648
```

- Example 2:

```
int value = -2147483648 - 1;
//The value will actually be 2147483647
```

- Java does not detect overflow errors so you have to be careful.

Numerical Input

- The Scanner class provides various methods for reading different numeric data types.

- Always use the method which reads the correct data type.
  - Entering an incorrect data type results in a Runtime Error.

Method	Description
`nextByte()`	reads an integer of the byte type.
`nextShort()`	reads an integer of the short type.
`nextInt()`	reads an integer of the int type.
`nextLong()`	reads an integer of the long type.
`nextFloat()`	reads a number of the float type.
`nextDouble()`	reads a number of the double type.

Numeric Operators

- Java has five numeric operators:
  - +   Addition
  - -   Subtraction
  - *   Multiplication
  - /   Division
  - % Modulus or Remainder Division

Integer vs Floating-Point Division

- If both operands of division (/) are integers then the result of the division will be an integer, the fractional part (decimal part) is truncated (cut off).

- Examples:

  5 / 2 = 2 instead of 2.5

  -5 / 2 = -2 instead of -2.5

- To do regular division one (or both) of the operands must be a floating-point number.

- Example: 5.0 / 2 or 5 / 2.0 = 2.5

Round-off Errors

- A ***round-off error (rounding error)***, is the difference between the calculated approximation of a number and its exact mathematical value.

- Example: 1/3 is approximately 0.333 with three decimal places, 0.3333333 with seven decimal places
  - the number of digits that can be stored in a variable is limited and round-off errors can occur.

- Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy.
  - Example:

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
//displays 0.5000000000000001, not 0.5
System.out.println(1.0 - 0.9);
//displays 0.09999999999999998, not 0.1
```

The Modulo (Mod) Operator (%)

- The modulo operator (%) gives the remainder after division.
  - operand on left is the ***dividend***
  - operand on the right is the ***divisor***

- Can be used with positive or negative numbers but **the result is only negative if the left had side of the operator is negative**.

- Examples:

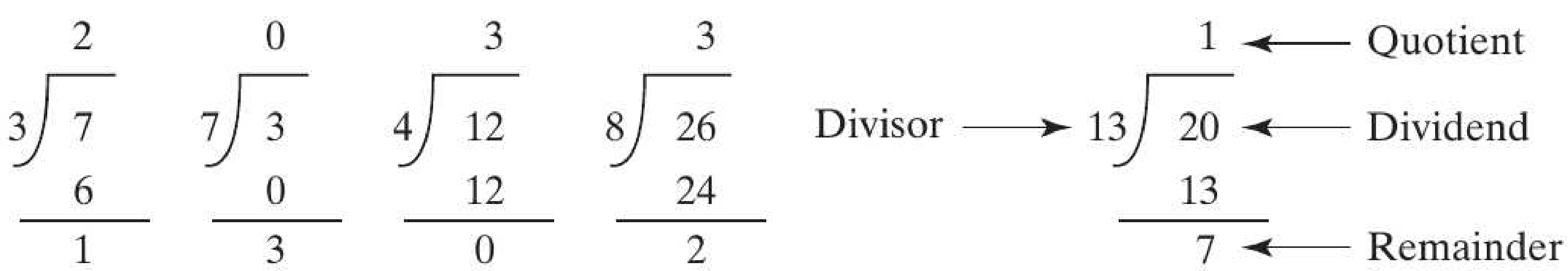
  7 % 3 = 1

  3 % 7 = 3

  12 % 4 = 0

  -7 % 3 = -1

  3 % -7 = 3



The Modulo (Mod) Operator (%)

- Can have many uses, two of which are:
- Determining if a number is even or odd.
  - an even number % 2 is always 0
  - an odd number % 2 is always 1.
- Determining if a number is evenly divisible by another number.
  - If a number % another number is 0 then it is evenly divisible otherwise it is not.

Example: SepDigits.java

- Task: Given a 3 digit number, say 342, separate the three digits.
  - We can do this using a combination of division and modulo.

- The sample output is:
  - The first digit is: 3
  - The second digit is: 4
  - The third digit is: 2

Example: SepDigits.java

```
public class SepDigits {
   public static void main(String[] args) {
      int num = 342;
      int first, second, third;

      first = num / 100;  //get the first digit

      second = (num % 100) / 10; //get the second digit

      third = num % 10; // get the third digit

      System.out.println("The first digit is: "+ first);
      System.out.println("The second digit is: "+ second);
      System.out.println("The third digit is: "+ third);
   }
}
```

Powers and Roots

- Java does not have built in operators for powers and roots. You have to use methods from the **`Math`** class.

- Calculating Powers:
  - The **`Math.pow(a, b)`** method in the **`Math`** class can be used to compute a^b
  - Example: Compute 2^3

```
double answer = Math.pow(2, 3);
```

- Calculating Roots:
  - The **`Math.sqrt(x)`** method in the Math class can be used to compute the square root of a number
  - Example: Compute the square root of 4.

```
double answer = Math.sqrt(4);
```

Numeric Literals

Numeric Literals

- A ***literal*** is a value that is typed directly in the source code.

- For example, **`34`** and **`0.305`** are literals in the following:

```
int numberOfYears = 34;
double weight = 0.305;
```

Integer Literals

- Integer literals can be assigned to integer variables as long as they can fit into the variable.

- A compilation error would occur if the literal were too large for the variable to hold
  - Example: **`byte b = 1000;`** would cause a compilation error since 1000 cannot be stored in a variable of the byte type

Integer Literals

- Integer literals are assumed to be of the `int` data type
- To assign an integer literal to a long data type you have to append an `L` or l to the end of the literal
  - Example: `long variableName = 2147483648L`
- Note: `L` is preferred because `l` (lowercase `L`) can be confused with `1` (the number one)

Floating-Point Literals

- Floating-Point Literals are numeric literals written directly in the source code that contain decimal points.

- By default they are treated as a **`double`** type value
  - Example: 5.0 is considered a **`double`** value not a **`float`** value.

- A floating-point literal can be made a **`float`** type by appending an **`F`** or **`f`** to the end of the number and can also be made a **`double`** type by appending a **`D`** or **`d`** to the end of the number.
  - Example:

    100.2f or 100.2F can be used for **`float`** numbers

    100.2d or 100.2D can be used for **`double`** numbers

Evaluating Expressions & Operator Precedence

Evaluating Expressions and Operator Precedence

- Parentheses:
  - Parentheses can change the order in which arithmetic operations are performed

- Examples:

```
int price = (cost + tax) * discount
int price = cost + (tax * discount)
```

- Without parentheses, an expressions is evaluated according to the rules of precedence.

Order of Operations

- 1. Operations in parenthesis are evaluated first and parenthesis can be nested in which case innermost parentheses are evaluated first.

- 2. Multiplication, division, and modulo are evaluated next.
  - If an expression has several of these operator types they are applied left to right.
  - Multiplication, Division, and Mod operators have the same level of precedence

- 3. Addition and subtraction are evaluated next.
  - If an expression has several of these operator types, they are applied left to right.
  - Addition and Subtraction have the same level of precedence.

Sample Expressions

- Math expressions need to be translated into a Java format before they can be evaluated.

Ordinary Mathematical Expression	Java Expression (Preferred Form)	Equivalent Fully Parenthesized Java Expression
$rate^2 + delta$	`rate*rate + delta`	`(rate*rate) + delta`
$2(salary + bonus)$	`2*(salary + bonus)`	`2*(salary + bonus)`
$\frac{1}{time + 3\ mass}$	`1/(time + 3*mass)`	`1/(time + (3*mass))`
$\frac{a-7}{t+9v}$	`(a - 7)/(t + 9*v)`	`(a - 7)/(t + (9*v))`

Augmented Assignment Operators

Augmented Assignment Operators

- In an assignment statement, it's common that the current value of a variable is used, modified, and then reassigned back to the same variable
  - Example: `i = i + 8`
  - Adds the current value of **`i`** with **`8`** and assigns the result back to **`i`**

- We can combine the assignment and addition operators using a shorthand operator
  - Example: `i += 8`

    `+=` is called the addition assignment operator

- There is an augmented assignment operator for each of the five number operations:
  - +=, -=, *=, /=, %=

Increment and Decrement Operators

Increment and Decrement Operators

- The increment (++) and decrement (--) operators are shorthand operators used to increment or decrement a variable by 1.

- They can appear before the variable name (prefix increment or decrement) or after the variable name (postfix increment or decrement)
  - i.e. ++x or x++

- The operators will have different effects depending on the position if used in a statement.

Numeric Type Conversions (Casting)

Numeric Type Conversions

- A numeric value can always be assigned to a variable whose type supports a larger range of values without casting:

range increases

byte → short → int → long → float → double

- To assign a numeric value of a larger type to a smaller type you must cast the variable to the smaller type i.e. assigning a double to an int

- Note: Casting from a larger type to a smaller type will result in a loss of information and could lead to inaccurate results.

Numeric Type Conversions

- When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:
  - 1. If one of the operands is double, the other is converted into double.
  - 2. Otherwise, if one of the operands is float, the other is converted into float.
  - 3. Otherwise, if one of the operands is long, the other is converted into long.
  - 4. Otherwise, both operands are converted into int.

Numeric Type Conversions

- Implicit Casting:

  `double d = 3;` (type widening)

- Explicit Casting:

  `int i = (int)3.0;` (type narrowing)

  `int i = (int)3.9;` (Fraction part is truncated)

Numeric Type Conversions

- Casting does not change the variable being cast i.e. d is not changed after the following code:

```
double d = 4.5;
int i = (int)d; //i is 4, d is still 4.5
```

- The following code is still correct:

```
int sum = 0;
sum += 4.5; //sum is 4
```

- `sum += 4.5` is equivalent to `sum = (int)(sum + 4.5);`

References

- Liang, Chapter 02: Elementary Programming