CSS 142

Lecture 3

Arkady Retik aretik@uw.edu

TODAY'S CONTENT

1. **Recap**
2. **Assignments, statements, methods**
3. **Intro to classes and objects; String Class**
4. **Documentation, Coding Style**
5. **Hands-on: Class Activity**
6. **Reading | FoR:**
   - **reading Wedn: Chapter 2 (finish);**
   - **Next week Monday** Chapter 3 (**3.1, 3.2**)

FoR 2:

The order is from most common muddy question to least common muddy questions.

1. The printf() was overall confusing and understanding how to format using the **printf method**. i.e %.2f, %6.2f, %+.2f, %s, etc

2**. IndexOf(String**, startingIndex) and IndexOf() method was confusing

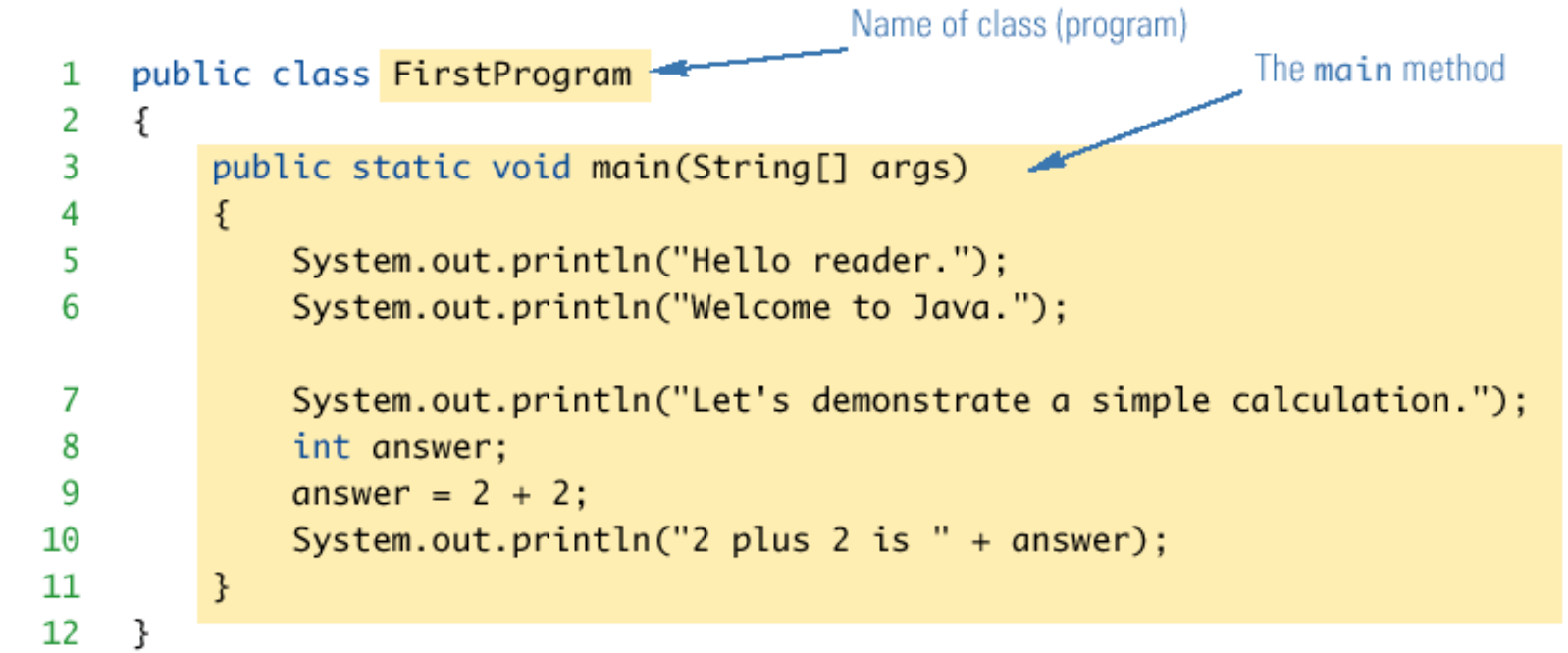
String name = “Mary, Mary quite contrary”
Name.indexOf(“Mary”, 1) ;
This return 6 and not 0 or why it returns 6 was asked.

3. **Escape sequences** and why do we use them was commonly asked too.

4. What is the difference between **classes, objects, and methods** was asked often.

Using internal input of data

Display 1.1 A Sample Java Program




```
1  public class FirstProgram
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello reader.");
6          System.out.println("Welcome to Java.");
7          System.out.println("Let's demonstrate a simple calculation.");
8          int answer;
9          answer = 2 + 2;
10         System.out.println("2 plus 2 is " + answer);
11     }
12 }
```

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

Using external input of data

LISTING 1.1 A Sample Java Program

```
import java.util.Scanner;

public class FirstProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello out there.");
        System.out.println("I will add two numbers for you.");
        System.out.println("Enter two whole numbers on a line:");

        int n1, n2;

        Scanner keyboard = new Scanner(System.in);
        n1 = keyboard.nextInt();
        n2 = keyboard.nextInt();

        System.out.println("The sum of those two numbers is");
        System.out.println(n1 + n2);
    }
}
```

Gets the `Scanner` *class from the package (library)* `java.util`

Name of the class—your choice

Sends output to screen

Says that `n1` *and* `n2` *are variables that hold integers (whole numbers)*

Readies the program for keyboard input

Reads one whole number from the keyboard

Sample Screen Output

```
Hello out there.
I will add two numbers for you.
Enter two whole numbers on a line:
12 30
The sum of those two numbers is
42
```

Programming in Java

Fundamental Building Blocks of Programs

DATA

- **variables**

  Memory location/container

- **types**

  kind of data

INSTRUCTIONS

- **control structures**

  Loops and branches

- **subroutines**

  methods & functions

OOP *structure/tools to deal with complexity*

Syntax and Semantics

- *the syntax describes **how** you write a program and*
- *the semantics describes **what** happens when you run the program.*

Programming in Java: Data, Classes, Objects and Methods

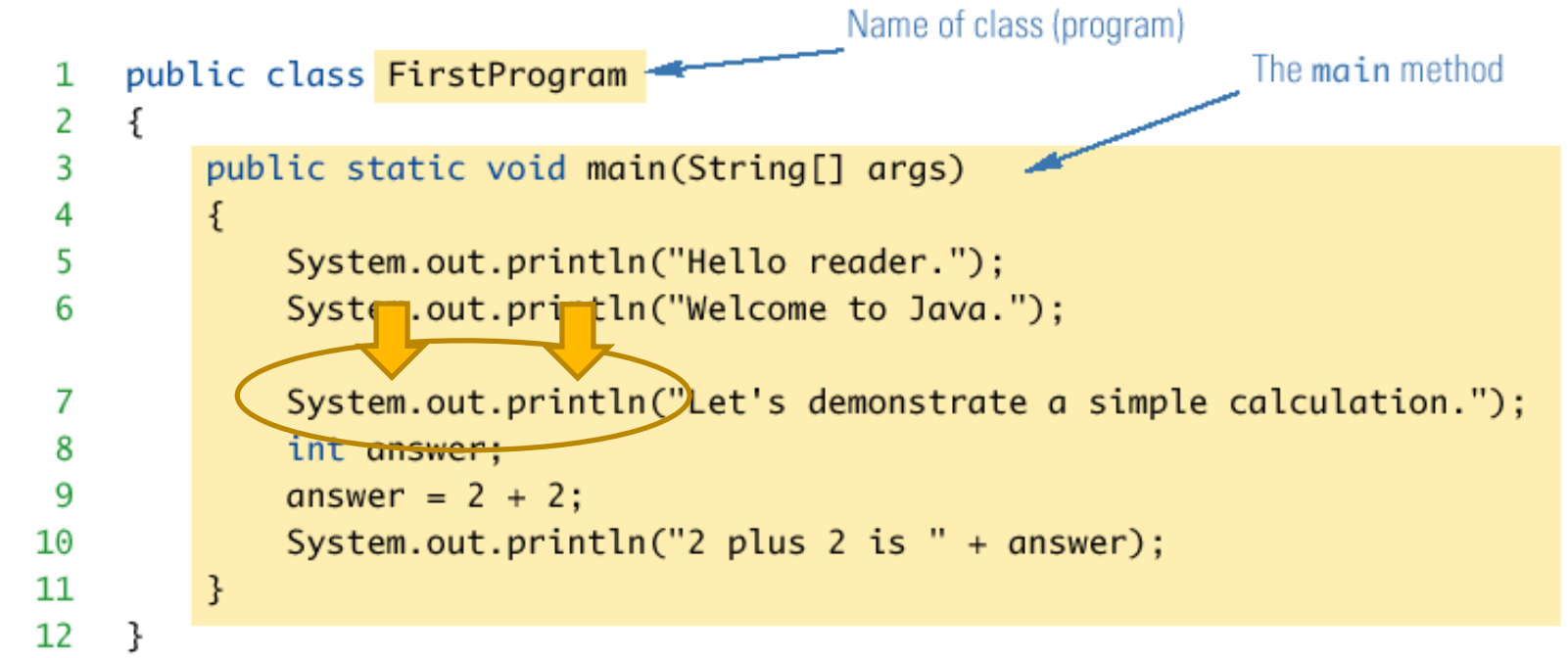
- Java is an *object-oriented programming (**OOP)*** language
  - Programming methodology that views a program as consisting of ***objects*** that interact with one another by means of **actions** (called ***methods***)
  - Objects of the same kind are said to have the same ***type*** or be in the same ***class***

- **In 142 mostly:**
  - One class (i.e. one file)
  - Focus on Data (primitive) and Methods

Java Application Programs

- There are two types of Java programs: ***applications*** and ***applets***
- A Java ***application*** *program* or "regular" Java program is a class with a method named **`main`**
  - When a Java application program is run, the ***run-time system*** automatically invokes the method named **`main`**
  - All Java application programs start with the **`main`** method

Display 1.1 A Sample Java Program




```
1  public class FirstProgram
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello reader.");
6          System.out.println("Welcome to Java.");
7          System.out.println("Let's demonstrate a simple calculation.");
8          int answer;
9          answer = 2 + 2;
10         System.out.println("2 plus 2 is " + answer);
11     }
12 }
```

demo

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

`System.out.println`

- Java programs work by having things called *objects* perform actions
  - `System.out`: an **object** used for sending output to the screen
- The actions performed by an object are called ***methods***
  - `println`: the method or action that the `System.out` object performs

`System.out.println`

- ***Invoking*** or ***calling*** a method: When an object performs an action using a method
  - Also called *sending a message* to the object
  - **Method invocation syntax** (in order): an object, a dot (period), the method name, and a pair of parentheses
  - **Arguments**: Zero or more pieces of information needed by the method that are placed inside the parentheses

```
System.out.println("This is an argument");
```

Variable declarations

- Variable declarations in Java are similar to those in other programming languages
  - Simply give the type of the variable followed by its name and a semicolon

```
int answer;
```

```
System.out.println("Hello reader.");
System.out.println("Welcome to Java.");

System.out.println("Let's demonstrate a simple calculation.");
int answer;
answer = 2 + 2;
System.out.println("2 plus 2 is " + answer);
```

Who calculate **`answer`** and how?

Using = , + (and ==)*

- In Java, the equal sign (=) is used as the ***assignment*** *operator*
  - The variable on the left side of the assignment operator is *assigned the value* of the expression on the right side of the assignment operator

```
answer = 2 + 2;
```

- In Java, the plus sign (+) can be used to denote addition (as above) or *concatenation*
  - Using +, two strings can be connected together

```
System.out.println("2 plus 2 is " + answer);
```

**** == is used for "equal"***

Examples

- int answer = 10+12; //addition

- int a = 8;
- int b = 14;
- answer = a+b; //addition

What's missing here?

- String first = "CSS";
- String last = "142";
- String name = first + " " + last; //concatenation

- System.out.println("Your time for " + name + " per week is " + answer);

Computer Language Levels

- *High-level language*: A language that people can read, write, and understand
  - A program written in a high-level language must be translated into a language that can be understood by a computer before it can be run
- *Machine language*: A language that a computer can understand
- *Low-level language*: Machine language or any language similar to machine language
- *Compiler*: A program that translates a high-level language program into an equivalent low-level language program
  - This translation process is called ***compiling***

Byte-Code and the Java Virtual Machine

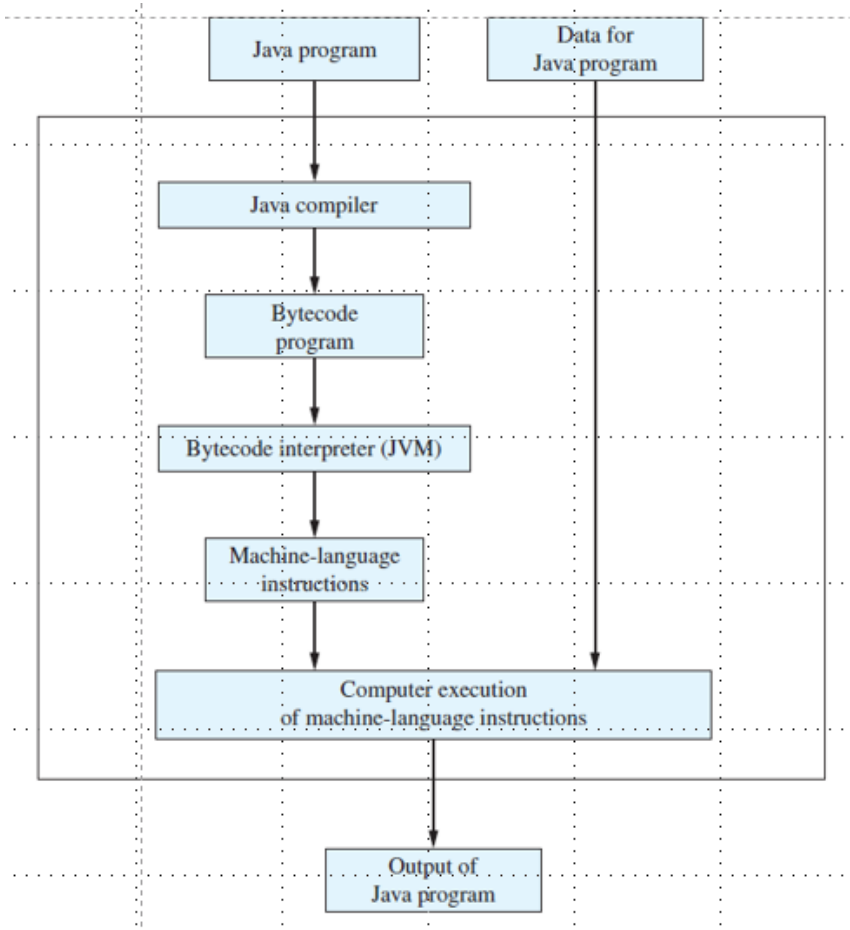
- The compilers for most programming languages translate high-level programs directly into the machine language for a particular computer
  - Since different computers have different machine languages, a different compiler is needed for each one
- In contrast, the Java compiler translates Java programs into ***byte-code***, a machine language for a fictitious computer called the *Java Virtual Machine*
  - Once compiled to *byte-code*, a Java program can be used on any computer, making it very **portable**
    - Can be ‘ported’ to a different system, device, etc.

Byte-Code and the Java Virtual Machine

- ***Interpreter:*** The program that translates a program written in Java byte-code into the machine language for a particular computer when a Java program is executed
  - The interpreter translates and immediately executes each byte-code instruction, one after another
  - Translating byte-code into machine code is relatively easy compared to the initial compilation step

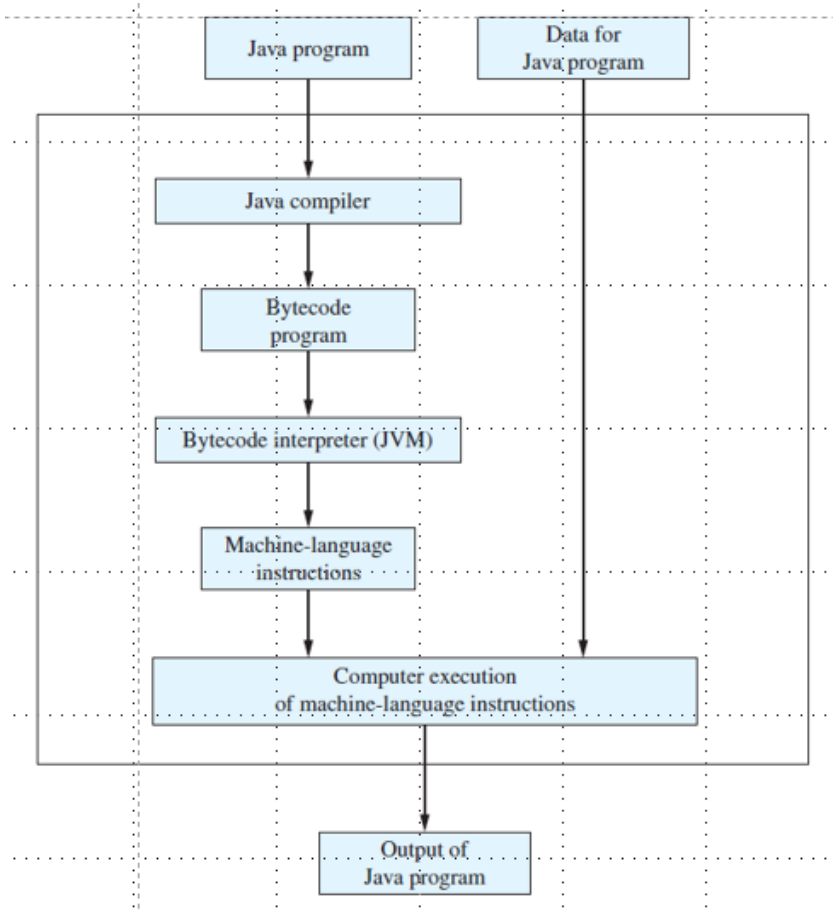
Program terminology

- *Code*: A program or a part of a program [what's *bits, binaries ?*]
- *Source code* (or *source program*): A program written in a high-level language such as Java
  - Something you can edit / modify
  - The input to the compiler program
- *Object code*: The translated low-level program
  - The output from the compiler program, e.g., Java byte-code
  - In the case of Java byte-code, the input to the Java byte-code interpreter



Class Loader

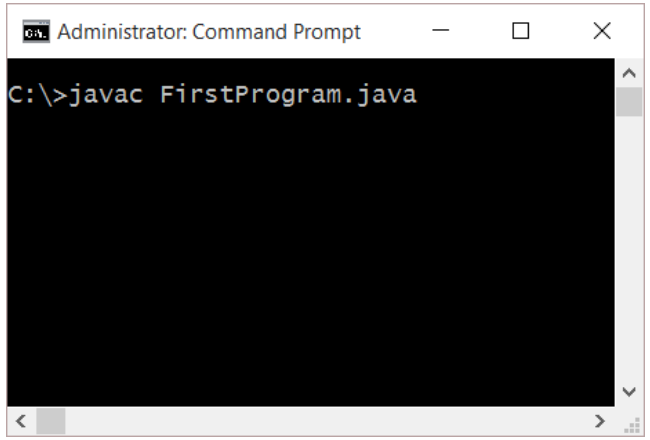
- Java programs are divided into smaller parts called *classes*
  - Each class definition is normally in a separate file and compiled separately
- *Class Loader*: A program that connects the byte-code of the classes needed to run a Java program
  - In other programming languages, the corresponding program is called a *linker*



Lab 1: how many (and what) files BlueJ produced?

Compiling a Java Program or Class

- Each class definition must be in a file whose name is the same as the class name followed by **`.java`**
  - The class **`FirstProgram`** must be in a file named **`FirstProgram.java`**
- Each class is compiled with the command **`javac`** followed by the name of the file in which the class resides
  - **`javac FirstProgram.java`**
  - The result is a byte-code program whose filename is the same as the class name followed by **`.class`**
  - **`FirstProgram.class`**

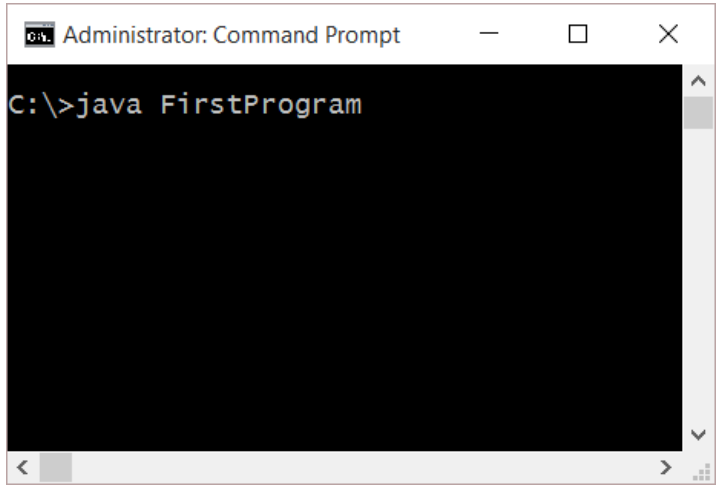


Running a Java Program

- A Java program can be given the *run command* (**`java`**) after all its classes have been compiled
  - Only run the class that contains the **`main`** method (the system will automatically load and run the other classes, if any)
  - The **`main`** method begins with the line:
  - **`public static void main(String[ ] args)`**
- Follow the run command by the name of the class only (no **`.java`** or **`.class`** extension)

```
C:\> java FirstProgram
```

Command line prompt



Syntax and Semantics

- *Syntax*: The arrangement of words and punctuations that are legal in a language, the *grammar rules* of a language

- *Semantics*: The meaning of things written while following the syntax rules of a language [we often call semantic ***logic***]

From our Affinity Exercise:

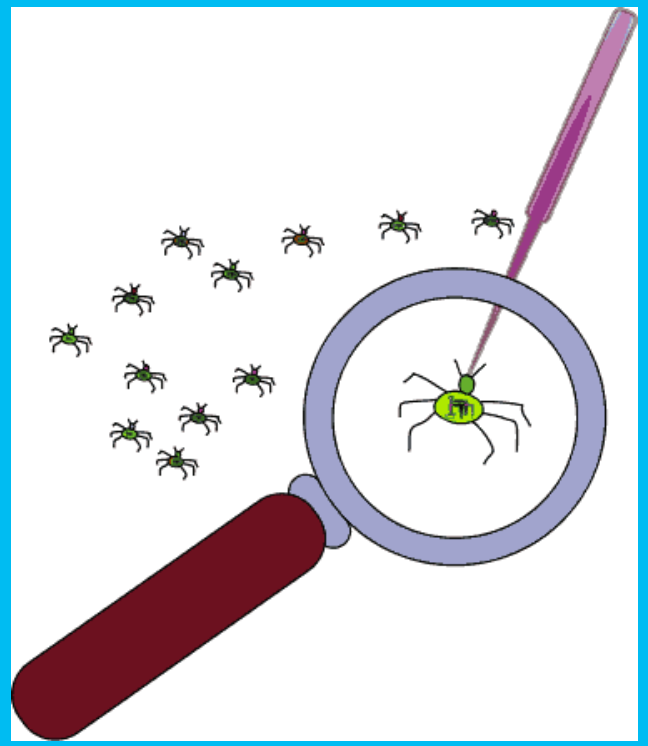
“I want good mark” .

Find one grammar and one semantic problem

Error Messages

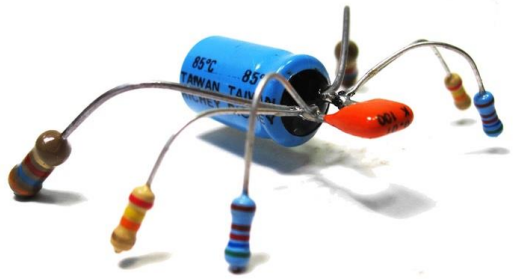
- ***Bug*:** A mistake in a program
  - The process of eliminating bugs is called ***debugging***
- ***Syntax error****:* A grammatical mistake in a program
  - The compiler can detect these errors, and will output an error message saying what it thinks the error is, and where it thinks the error is

INTRO to DEBUGGING

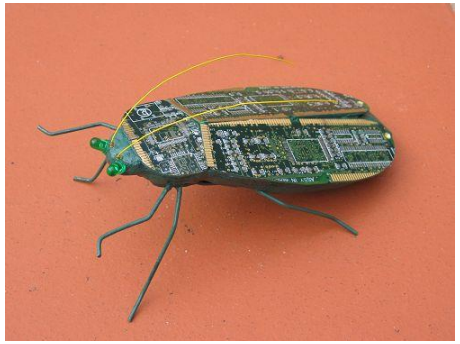


Introduction

Program design -> { coding -> testing | debugging }



- No one has find a way to avoid mistakes (errors/bugs)
  - Even Savitch;
- Debugging (**elimination of mistakes**) is a skill that require practice to master
- Don’t be afraid of buds -> learn from them!



Types of Errors

Syntax errors (Grammar)

- Compiler will always detect syntax error but not what's wrong
- Compiler is not good telling where the real error in case of multiple errors

Run-time errors

- Detected when your program is run; it will inputs an error message
- The error messages may not be easy to understand

Logic Errors (Semantics)

- When your program compiles without error, you are still not done.
- You have to test the program to make sure it works correctly.

General Guidelines and Rules of Thumb

Avoid lots of errors by following some basic programming guidelines:

- Never type a "**{**" without typing the matching "**}**".
  - Then go back and fill in the statements between the braces. A missing or extra brace can be one of the hardest errors to find in a large program.
- Always, always **indent your program** nicely. If you change the program, change the indentation to match. It's worth the trouble.
- Use a **consistent naming scheme.**
- When the compiler gives multiple error messages, don't try to fix the second error message from the compiler until **you've fixed the first one**.

General Guidelines and Rules of Thumb /continue

Maybe the best advice is:

❖ Take the time to **understand the error** before you try to fix it -> Programming is not an experimental science.

- Often, it's a problem just to find the part of the program that contains the error. The debugger allows you to set **"breakpoints"** in your program. A breakpoint is a point in the program where the debugger will pause the program so you can look at the values of the program's variables.
- The debugger will also let you execute your program **one line at a time**, so that you can watch what happens in detail once you know the general area in the program where the bug is lurking.

General Guidelines and Rules of Thumb /continue

- A more traditional approach to debugging is **to insert debugging statements** into your program. These are output statements that print out information about the state of the program. Typically, a debugging statement would say something like

  System.out.println("At start of while loop, n = " + n);

- You need to be able to tell from the output where in your program the output is coming from, and you want to know the value of important variables. Sometimes, you will find that the computer isn't even getting to a part of the program that you think it should be executing.
- Remember that the goal is **to find the first point** in the program where the state is not what you expect it to be. **That's where the bug is.**

Java Syntax

Identifiers

- ***Identifier***: The name of a variable or other item (class, method, object, etc.) defined in a program
  - A Java identifier **must not start with a digit**, and all the characters must be letters, digits, or the underscore symbol
  - Java identifiers can theoretically be of any length
  - Java is a **case-sensitive** language: `Score`, `score`, and `SCORE` are the names of three different variables

Identifiers

- **Keywords** and **Reserved** words: Identifiers that have a predefined meaning in Java
  - Do not use them to name anything else

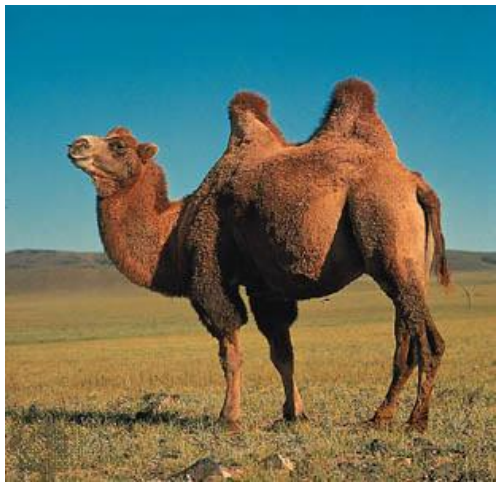
```
public    class    void    static
```

- **Predefined identifiers**: Identifiers that are defined in libraries required by the Java language standard
  - Although they can be redefined, this could be confusing and dangerous if doing so would change their standard meaning

```
System    String    println
```

Naming Conventions

- Start the names of **variables, methods, and objects** with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters
  - `topSpeed` `bankRate1` `timeOfArrival` } Camel Style
- Start the **names of classes** with an uppercase letter and, otherwise, adhere to the rules above
  - `FirstProgram` `MyClass` `String`



Variable Declarations

- Every variable in a Java program must be *declared* before it is used
  - A variable declaration tells the compiler what kind of data (**type**) will be stored in the variable
  - The type of the variable is followed by one or more variable names **separated by commas,** and **terminated with a semicolon**
  - Variables are typically **declared just before they are used** or at the start of a block (indicated by an opening brace **{**)
  - Basic types in Java are called ***primitive*** *types*
    - `int numberOfBeans;`
    - `double oneWeight, totalWeight;`

Why need to declare?

Memory will be allocated accordingly

Display 1.2 Primitive Types

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
`boolean`	`true` or `false`	1 byte	not applicable
`char`	single character (Unicode)	2 bytes	all Unicode characters
`byte`	integer	1 byte	−128 to 127
`short`	integer	2 bytes	−32768 to 32767
`int`	integer	4 bytes	−2147483648 to 2147483647
`long`	integer	8 bytes	−9223372036854775808 to 9223372036854775807
`float`	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
`double`	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

What's the difference?

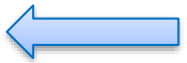
Assignment Statements With Primitive Types

- In Java, the assignment statement is used to change the value of a variable
  - The equal sign (=) is used as the **assignment operator**
  - An assignment statement consists of a variable on the left side of the operator, and an *expression* on the right side of the operator
    - `Variable = Expression;`
  - An *expression* consists of a variable, number, or mix of variables, numbers, operators, and/or method invocations
    - `temperatureToday = 70.5;`
    - `count = numberOfDays;`

Assignment Statements With Primitive Types

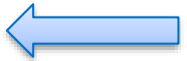
- When an assignment statement is executed, the expression is first **evaluated**, and then the **variable** on the left-hand side of the equal sign **is set** equal to the value of the expression

```
distance = rate * time;
```



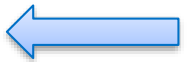
- Note that a variable can occur on both sides of the assignment operator

```
count = count + 2;
```



- The assignment operator is automatically executed from **right-to-left**, so assignment statements can be chained

```
number2 = number1 = 3;
```



Tip: Initialize Variables

- A variable that has been declared but that has not yet been given a value by some means is said to be ***uninitialized***
- In certain cases an uninitialized variable is given **a default value**
  - It is best not to rely on this
  - **Explicitly initialized variables** have the added benefit of improving program clarity

Tip: Initialize Variables

- The declaration of a variable can be combined with its initialization via an assignment statement
  - `int count = 0;`
  - `double distance = 55 * .5;`
  - `char grade = 'A';`
  - Note that some variables can be initialized and others can remain uninitialized in the same declaration
    - `int initialCount = 50, finalCount;`


Shorthand Assignment Statements

- Shorthand assignment notation combines the *assignment operator* (**=**) and an ***arithmetic op****erator* (some examples of what **`Op`** can be are **`+`**, **`-`**, **`*`**, **`/`**, or **`%`**)
- It is used to change the value of a variable by adding, subtracting, multiplying, or dividing by a specified value
- The general form is

```
Variable Op = Expression
```

  - which is equivalent to

```
Variable = Variable Op (Expression)
```

  - The **`Expression`** can be another variable, a constant, or a more complicated expression

Examples of Shorthand Assignment Statements

Example:	Equivalent To:
count += 3;	count = count + 3;
sum -= discount;	sum = sum – discount;
bonus *= 2;	bonus = bonus * 2;
time /= rushFactor;	time = time / rushFactor;
change %= 55;	change = change % 55;
amount *= count1 + count2;	amount = amount * (count1 + count2);

Assignment Compatibility

- In general, the value of one type **cannot** be stored in a variable of another type
  - `int intVariable = 2.99; //Illegal`
  - The above example results in a type mismatch because a `double` value cannot be stored in an `int` variable
- However, there are exceptions to this
  - `double doubleVariable = 2;`
  - For example, an `int` value can be stored in a `double` type

Why is that?

Size of the memory allocation

Assignment Compatibility and Type Cast

- More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it
  - `byte→short→int→long→float→double`
  - `char`
  - Note that as your move down the list from left to right, the range of allowed values for the types becomes larger
- An explicit ***type cast*** is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., `double` to `int`)
- **Note** that in Java an `int` **cannot** be assigned to a variable of type `boolean`, nor can a `boolean` be assigned to a variable of type `int`

In some languages
TRUE==1;
FALSE==0;

Constants

- ***Constant*** (or ***literal***): An item in Java which has one specific value that **cannot change**
  - Constants of an integer type may not be written with a decimal point (e.g., `10`)
  - Constants of a floating-point type can be written in ordinary decimal fraction form (e.g., `367000.0` or `0.000589`)
  - Constant of a floating-point type can also be written in *scientific* (or *floating-point*) *notation* (e.g., `3.67e5` or `5.89e-4`)
    - Note that the number before the `e` may contain a decimal point, but the number after the `e` may not

Constants

- Constants of type **`char`** are expressed by placing a **single character** in single quotes (e.g., **`'Z'`**)
- Constants for strings of characters are enclosed by **double quotes** (e.g., **`"Welcome to Java"`**)
- There are only two **`boolean`** type constants, **`true`** and **`false`**
  - Note that they **must be spelled with all lowercase letters**

Arithmetic Operators and Expressions

Arithmetic Operators and Expressions

- As in most languages, ***expressions*** can be formed in Java using **variables**, **constants**, and **arithmetic operators**
  - These operators are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo, remainder)
  - An expression can be used anyplace it is legal to use a value of the type produced by the expression

Arithmetic Operators and Expressions

- If an arithmetic operator is combined with **`int`** operands, then the resulting type is **`int`**
- If an arithmetic operator is combined with one or two **`double`** operands, then the resulting type is **`double`**
- If different types are combined in an expression, then the resulting type is the right-most type on the following list that is found within the expression
  - **`byte→short→int→long→float→double`**
  - **`char`** 
  - **Exception:** If the type produced should be **`byte`** or **`short`** (according to the rules above), then the type produced will actually be an **`int`**

Parentheses and Precedence Rules

- An expression can be *fully parenthesized* in order to specify exactly what subexpressions are combined with each operator
- If some or all of the parentheses in an expression are omitted, Java will follow ***precedence*** rules to determine, in effect, where to place them
  - However, it's best (and sometimes necessary) to include them

Precedence Rules

Display 1.3 Precedence Rules

Highest Precedence

First: the unary operators: +, −, ++, −−, and !

Second: the binary arithmetic operators: *, /, and %

Third: the binary arithmetic operators: + and −

Lowest Precedence

Precedence and Associativity Rules

- When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence

  `base + rate * hours` is evaluated as

  `base + (rate * hours)`

- When two operations have equal precedence, the order of operations is determined by *associativity* rules

Precedence and Associativity Rules

- Unary operators of equal precedence are grouped right-to-left

  `+-+rate` is evaluated as `+(-(+rate))`

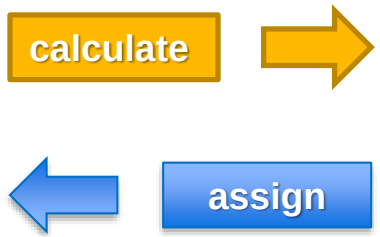
- Binary operators of equal precedence are grouped left-to-right

  `base + rate + hours` is evaluated as

  `(base + rate) + hours`

- Exception: A string of **assignmen**t operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`



Pitfall: Round-Off Errors in Floating-Point Numbers

- Floating point numbers are only approximate quantities
  - Mathematically, the floating-point number 1.0/3.0 is equal to 0.3333333 . . .
  - A computer has a finite amount of storage space
    - It may store 1.0/3.0 as something like 0.3333333333, which is slightly smaller than one-third
  - Computers actually store numbers in binary notation, but the consequences are the same: floating-point numbers may lose accuracy

What does this means for us?

Integer and Floating-Point Division

- When one or both operands are a floating-point type, division results in a floating-point type
  - `15.0/2` evaluates to `7.5`
- When both operands are integer types, division results in an integer type
  - Any fractional part is discarded
  - The number is not rounded
    - `15/2` evaluates to `7`
- Be careful to make at least one of the operands a floating-point type if the fractional portion is needed

The Modulo Operator: `%`

- The Module (or Mod) `%` operator is used with operands of type **`int`** to recover the information lost after performing integer division
  - **`15/2`** evaluates to the quotient **`7`**
  - **`15%2`** evaluates to the remainder **`1`**
- The `%` operator can be used to count by 2's, 3's, or any other number
  - To count by twos, perform the operation **`number`** `%` **`2`**, and when the result is **`0`**, **`number`** is even

Type Casting

- A ***type cast*** takes a value of one type and produces a value of another type with an "equivalent" value
  - If `n` and `m` are integers to be divided, and the fractional portion of the result must be preserved, at least one of the two must be type cast to a floating-point type **before** the division operation is performed
    - `double anws = n / (double)m;`
  - Note that the desired type is placed inside parentheses immediately in front of the variable to be cast
  - Note also that the type and value of the variable to be cast does not change
    - i.e. `m` will remain to be integer

More Details About Type Casting

- When type casting from a floating-point to an integer type, the number is **truncated**, not rounded
  - `(int)2.9` evaluates to `2`, not `3`
- When the value of an integer type is assigned to a variable of a floating-point type, Java performs an automatic type cast called a *type coercion*
  - `double d = 5;`
- In contrast, it is illegal to place a `double` value into an `int` variable without an explicit type cast
  - `int i = 5.5; // Illegal`
  - `int i = (int)5.5 // Correct`

Increment and Decrement Operators

- The *increment operator* (`++`) adds one to the value of a variable
  - If `n` is equal to `2`, then `n++` or `++n` will change the value of `n` to `3`
- The *decrement operator* (`--`) subtracts one from the value of a variable
  - If `n` is equal to `4`, then `n--` or `--n` will change the value of `n` to `3`

Increment and Decrement Operators

- When either operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
  - If `n` is equal to `2`, then `2*(++n)` evaluates to `6`
- When either operator follows its variable, and is part of an expression, then the expression is evaluated **using the original value of the variable**, and only then is the variable value changed
  - If `n` is equal to `2`, then `2*(n++)` evaluates to `4`

Appendix 2
in Absolute Java

Precedence and Associativity Rules 2

PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation: `.`, `[]`, `()`	Left to right
`++` (postfix, as in `x++`), `--` (postfix)	Right to left
The unary operators: `+`, `-`, `++` (prefix, as in `++x`), `--` (prefix), `!`, `~` (bitwise complement)[1]	Right to left
`new` and type casts (*Type*)	Right to left
The binary operators `*`, `/`, `%`	Left to right
The binary operators `+`, `-`	Left to right
The binary operators `<<`, `>>`, `>>>` (shift operators)[1]	Left to right
The binary operators `<`, `>`, `<=`, `>=`, `instanceof`	Left to right
The binary operators `==`, `!=`	Left to right
The binary operator `&`	Left to right
The binary operator `^` (exclusive or)[1]	Left to right
The binary operator `|`	Left to right
The binary operator `&&`	Left to right
The binary operator `||`	Left to right
The ternary operator (conditional operator) `?:`	Right to left
The assignment operators `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`	Right to left

[1] Not discussed in this book.

Hands-on: Class Activity

Will contribute to your final grade:

10% Activities and Participation

25% homework

Hands-on: Activity 1

Name/s:

Date:

Instructions: In pairs, work on the following problems **using pencil and paper**.

Problem 1. Write a method that takes as input two integers and returns their sum.

Problem 2. Write a method that takes as input a double and prints it twice.

Problem 3. Write a main method with test calls to the methods you wrote for problems 1 and 2. Predict the output you would get from running your main.

Problem 4. What is the output produced by the following lines of program code?

```
char a, b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

Problem 5. What is the output produced by the following lines of program code? What would be difference if we would use –n instead of n (in line 4).

```
int n = 3;
n++;
System.out.println("n == " + n);
n;  //what will be difference if we use  –n;
System.out.println("n == " + n);
```

Next Wedn

- **Homework 1 is available.** Read it and ask questions
- **Read:** Savitch Chapter 2 and start Chapter 3 (3.1)