Programming and Software Development
COMP90041

Lecture 1

Introduction

NOTE: Some of the Material in these slides are adopted from the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 20-02-2021

- Who's teaching and how?
- What is this subject about?
- How much time will this take me?
- How and where do I get help?
- How does assessment work?
- What does academic conduct mean?

Learning Outcomes

Office hours, each **Monday**, **10-11am**

- Via ZOOM
- Email: udaya@unimelb.edu.au
- Please include COMP90041 in the subject line when sending an email.

Office hours, each **Thursday**, **11am to 12 pm**

- Via ZOOM
- thuan.pham@unimelb.edu.au

Lecturers

- Start in week 2
- via ZOOM
- 6 Lab sheets + consultations

- Zhuohan Xie (zhuohan.xie@unimelb.edu.au)
- Rahul Sharma (sharma1@student.unimelb.edu.au)
- Andrew Naughton (andrew.naughton@unimelb.edu.au)
- Zhe Wang (zoe.wang1@unimelb.edu.au)
- Dengke Sha (dengke.sha@unimelb.edu.au))
- Gaunli Liu (guanli.liu1@unimelb.edu.au)

Tutorials

Who's Udaya

- **Professor,**
- School of Computing and Information Systems, Melbourne School of Engineering, University of Melbourne.
- Web: https://people.eng.unimelb.edu.au/udaya/
- **Leader - Quantum Computing Research,**
- Senior Member, IEEE.

- Research Interests:
  - Trust and Privacy in Networks
  - Sequences for communication and security
  - Cryptography
  - Combinatorics
  - Theory of error correcting codes

Who's Thuan?

- **Lecturer in Cybersecurity**
  - Software Security — building automated security testing techniques (e.g., Fuzzing) to discovery software vulnerabilities
  - https://thuanpv.github.io
  - Twitter: @thuanpv_
- **Past experience**
  - Lecturer, Software architect, Research Fellow, Start-up (co-)founders
- My hometown: Hanoi, Vietnam
- Father of two kids :)

- Object-Oriented (OO) software development
  - Program design, implementation and testing
  - OO concept
    - classes
    - objects
    - encapsulation
    - inheritance
    - polymorphism
  - The Java programming language
  - Problem solving
    - data structures
    - algorithms

Subject Overview

- **Determinism**
- **Small actions have big effects**
- **Utterly logical**
- **Consistent, fair, and unequivocally unbiased**
- **A compiler is the most patient teacher you will ever meet**

Programming is Doing Magic

Week	Lecture	Tutorials	Assignments
1	Introduction		
2	Console I/O	Lab 1: Getting Started	
3	Flow Control	Lab 2: Basics of Java	
4	Classes I	Lab 3: User Input	
5	Classes II	Lab 4: Writing Java Classes	**Timed Quiz: Sep 4** Assignment 1 Release
N/a	Teaching Break		
6	Arrays and ArrayLists	Lab 5: Arrays	
7	Inheritance and Polymorphism	Assignment Consultation	**Assignment 1 due**
8	Interfaces and Exceptions	Lab 6: Inheritance	Assignment 2 Release
9	File I/O	Lab 7: Exception Handling	
10	Generics	Assignment Consultation	**Assignment 2 due** Final Project Release
11	Guest Lecture	Project Consultation	
12	Recap & Advanced Topics	Project Consultation	

Final Project due: TBC

Semester Overview

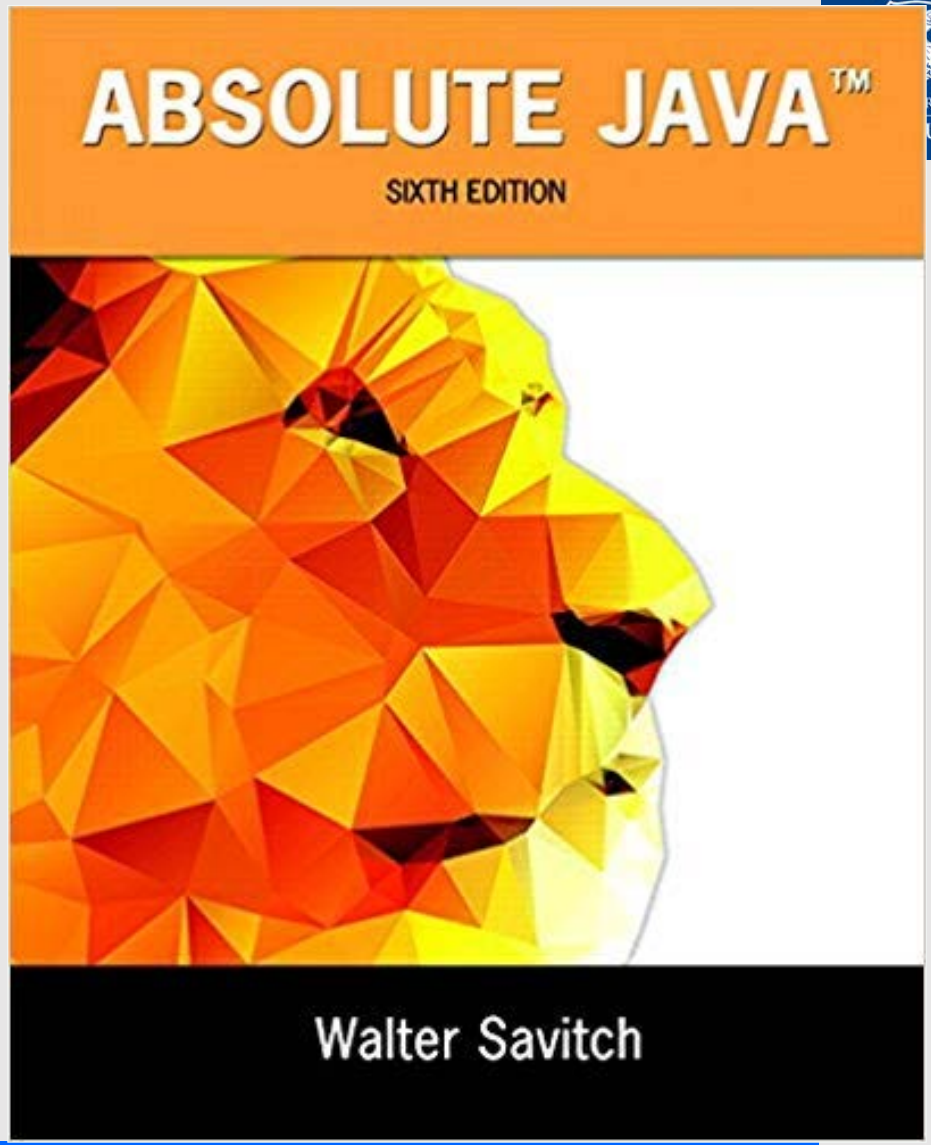
- Each Week we will have 2 Lectures,
- Lecture 1: Monday 5.15 pm.
- Lecture 2: Tuesday 5.15 pm.
- Please come prepared to Lectures by going through any material published for the week (slides, past presentations)
- I will be available for answering any questions after the lecture.
- We will have dedicated forum for discussing lecture contents.

Semester Overview

You can get it from the library

- Physically or

- Online



Textbook

- Two hours of lectures
- Two or three hours reading the textbook chapter
- One preparation hour for the tutorial
- One hour attending the tutorial.
- An hour of general review, perhaps in a study group.

- In total, around **seven or eight hours per week** is required, **starting immediately**.
- Make a study timetable for all activities. Then start following it.

Time Commitment

- This subject needs 1 or 2 student representatives
- Student reps act as a conduit for anonymous student feedback by email or in time set aside in one lecture
- Reps also report back to the SSLC (Student Staff Liaison Committee) meeting on how the class is going (and get a free lunch!)
- Help your classmates and get something to put on your CV
- Email me if you want to volunteer

Student Representatives

1. **Think**, read, and google it.
2. **Check Canvas** for content and announcements
3. Ask your **fellow students** for help and advice.
4. Consult the **Discussion Board** online. If your question has not been answered already, create an entry.
5. If the question still has not been answered, bring it up in your next **tutorial** session.
6. If that does not help, **email** us (Email Subject must start with COMP90041). We receive a high amount of emails.
7. If nothing else works, come to our office hours.

Where do I get help?

1. One **online Quiz**, 30 minutes duration within a 60 minute time window, taken via Canvas, closed book assessment (worth 10%).
2. **Two programming assignments**, worth 15% each.
3. A **final project** worth 60%.

To pass, a combined mark of 20/40 in the online quiz and programming assignments is required; a mark of 30/60 in the final exam; and an overall mark of at least 50/100 when all components are combined.

Assessment

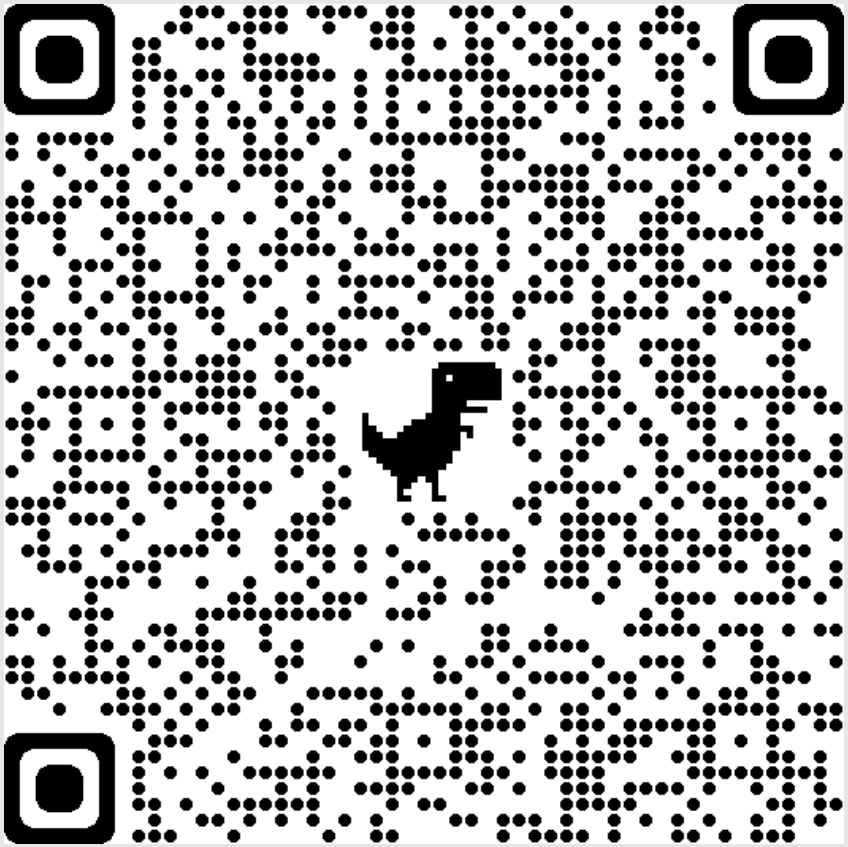
- Go to https://academicintegrity.unimelb.edu.au/ and inform yourself about the responsibilities you carry in an academic environment
- Enroll in https://catalog.lms.unimelb.edu.au/browse/communities/student-induction/courses/academic-integrity
- Be your best and carry these values forward
  - Plagiarism is no joke and not worth it

Academic Integrity

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections

Please fill in this microblog.



http://go.unimelb.edu.au/5o8i.

- Make sure you have friends in class

- Every time you enter a ZOOM over the next two weeks, turn on your video and mic, and introduce yourself to someone new, while you wait for the start of the class.

  - (we are starting the lecture 15min past the hour)

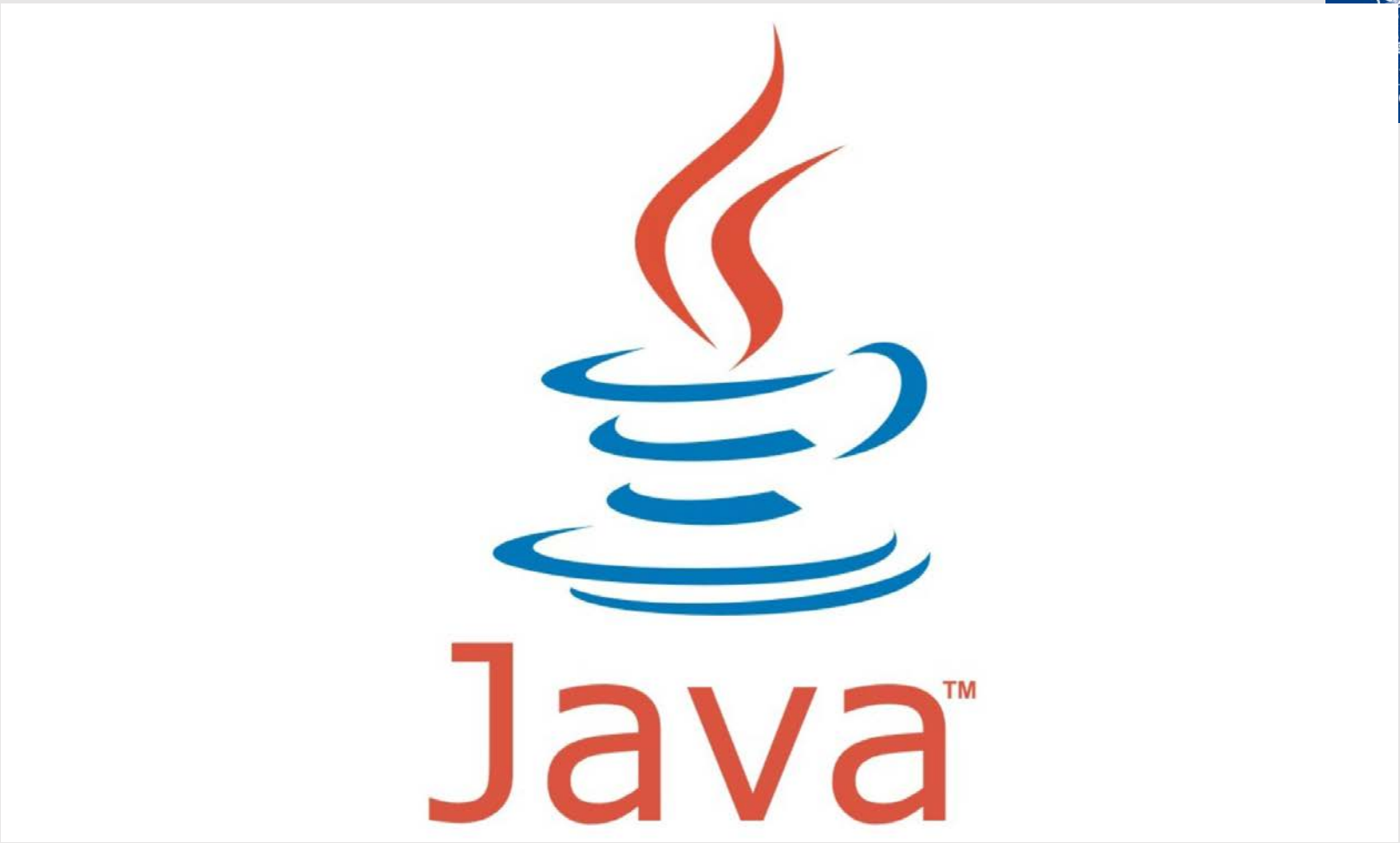
Make Friends!

- Things were challenging last year. The same situation may continue this year too!
- **Engagement and connectivity** are going to be problems to overcome. Please make a habit of actively contributing to tutorials, by leaving your video on, interacting with the tutors, and building networks with your classmates that extend beyond the formal sessions.
- It won't be as good as being on campus, but it should still be possible for you to **establish and maintain relationships** with a set of other students in the class. Please make the effort!

Make Friends!

- …

Take-Aways



Java Programming - Getting Started

- What is the structure of a Java program?

- How to write, compile, and run a Java program?
  - Using simple text editor (e.g., Vim)
  - Using an Integrated Development Environment (IDE) (e.g., Eclipse)

- Variable declaration & assignment

Outline

```
// display a friendly greeting
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

- Java **program** is made up of one or more **classes (Weeks 4, 5, 7, 8)**
- Java class is made up of zero or more **methods** and instance **variables (Weeks 1, 2)**
- Java method is made up of zero or more **statements**
- There are a few other things, like **comments**

First Java Program

System.out.println

- Prints something out to the console
- The "ln" part means "new line"
  - Next output will start a new line
- To print something without moving to another line, use
  - System.out.print

Standard output

- Every line of Java code must be in some text file. The filename must match the class name, including upper and lower casing, and always ends with .java
- Therefore the following class should be in a file called Hello.java

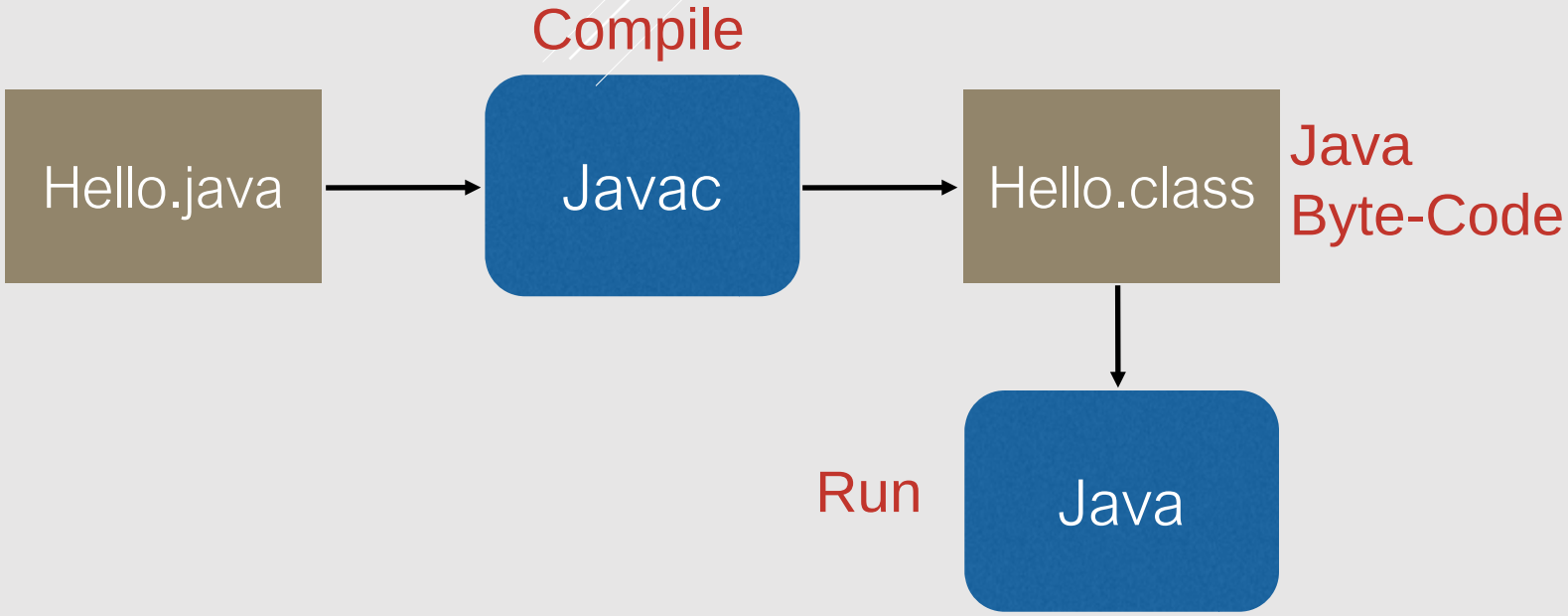
```
public class Hello
{
    // your code goes here
}
```

Files

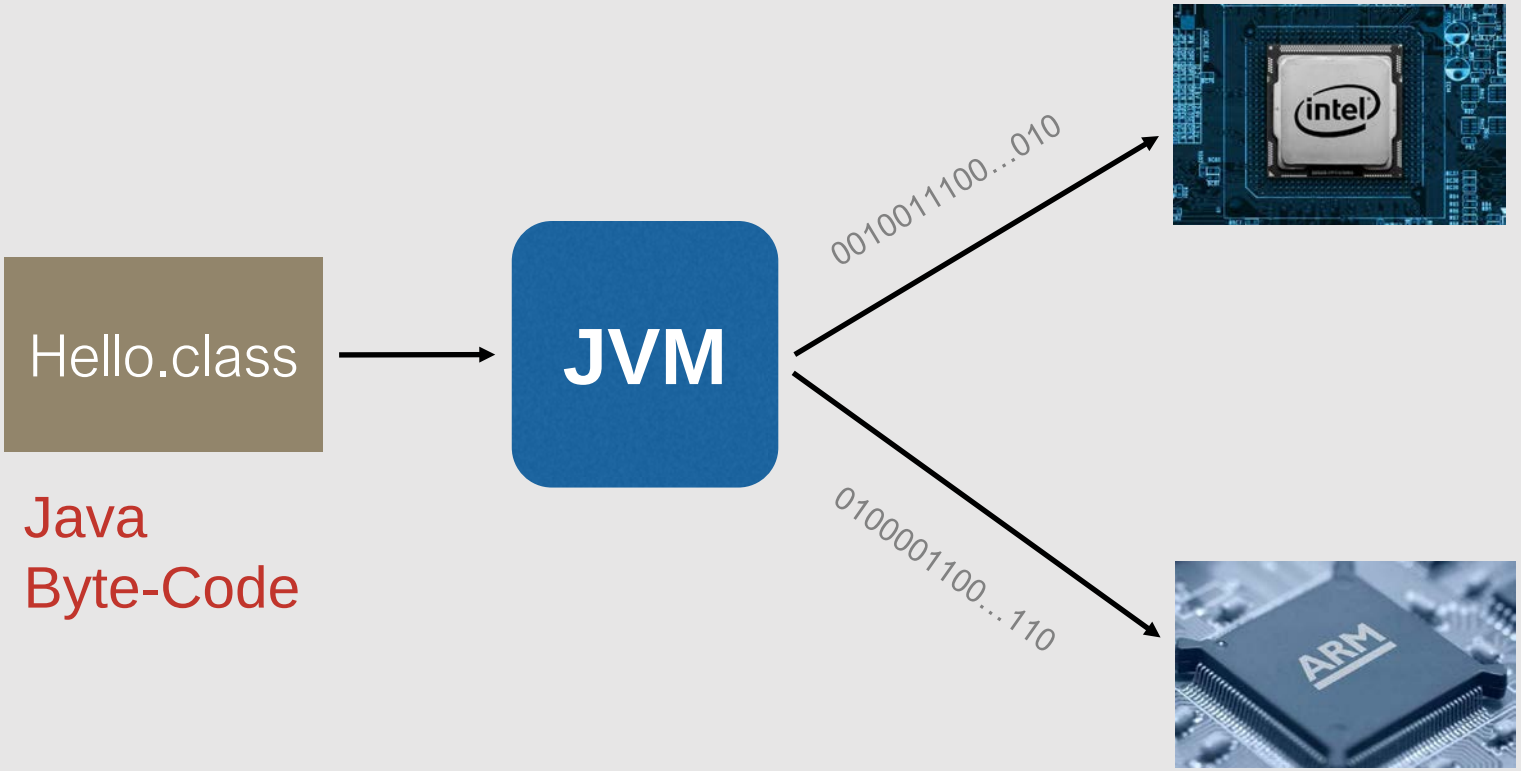
- Most people use an Integrated Development Environment (IDE) to create Java code
  - Popular IDEs include Eclipse and Netbeans
  - Both are free to download and use
  - Use whatever tools you like, even Notepad
- IDEs hide some boring details
  - But you still need to understand the details

Developing Code

- Before a java program can be run, it must first be compiled with the javac command
  - Checks the program obeys the rules of Java
  - Produces a .class file (if compiler passes)
- Once compiled, program is run using the java command



Compilation



Java Virtual Machine (JVM)

- Java applications run in a console
- Computer consoles originally looked like this

- This subject will focus on the console text input/output — No graphical user interfaces
- IDEs simulate console input/output

Execution

```
user$ javac Hello.java
user$ java Hello
Hello, World!
user$_
```

- **user$** represents my OS prompt; yours will be different
- Compile using **javac <filename.java>**
- If no errors detected, returns to prompt
- Run program with: **java <classname>** (not filename)
- Program output if shown, followed by next OS prompt; keyboard input may be needed

Command Line Execution

- What is the structure of a Java program?
- How to write, compile, and a Java program?
  - Using simple text editor (e.g., Vim)
  - Using an Integrated Development Environment (IDE) (e.g., Eclipse)
- Variable declaration & assignment

Outline

- Two parts of any program: code and data
- **Code** is the text of the program, what operations the program performs
- **Data** is what the code operates on
- Each datum (singular of data) has a type
- Three kinds of types: primitive, class, and array
  - We will cover class and array types later

Data

- Building blocks: all data are built from primitives
- Primitives can't be broken into smaller parts

Type	Bytes	Values
boolean	1	true, false
char	2	All Unicode chars (e.g., 'a', 'b', etc.)
byte	1	-2^7 to 2^7 -1 (-128 to 127)
short	2	-215 to 215 -1(-32768 to 32767)
int	4	-2^{31} to $2^{31} - 1 (\approx \pm 2 \times 10^9)$
long	8	-2^{63} to $2^{63} - 1 (\approx \pm 10^{19})$
float	4	$\approx \pm 3 \times 10^{38}$ (limited precision)
double	8	$\approx \pm 10^{308}$ (limited precision)

Primitive Types

- Variables have names and hold data
- Different values at different times
- Variable names begin with a letter, and follow with letters, digits, and underscores (_)
- Java naming convention for variable names is:
  - Begin with lower case letter
  - Follow with lower case, except
    - Capitalise first letter of each word in phrase
- E.g. height, windowHeight, tallestWindowHeight
- Best practice: make them descriptive, but not too long (clear abbreviations are OK)

Variable names

- Each variable *must be* declared, specifying its type
  - Type first, then variable name, then semicolon
  - E.g., int count; or boolean done;
- Variable ***must be*** assigned a value before being used
  - Specify variable first, then equal sign (=), then value followed by a semicolon
  - E.g., count = 1; or done = true;
- You can combine declaration with initial assignment
  - E.g., int count = 1; or boolean done = true;

Variable Declarations and Assignment

- Write Java classes in file named classname.java
- Compile and run the code using javac and java commands
- Variables hold values, can be assigned and reassigned
- Variables must be declared, with their types
- Variables must be initialised before being used

Summary

Programming and Software Development
COMP90041

Lecture 2

Console IO

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 20-02-2021

- Object Oriented Programming (Class vs Object)
- “Hello World” Java program
- Javac
- Java
- Variables
  - Primitive data types
  - Declaration and Assignment

Review

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Outline

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Outline

Type	Bytes	Values
`boolean`	1	`false`, `true`
`char`	2	All unicode characters (*e.g.*, `'a'`)
`byte`	1	-2^7 to $2^7 - 1$ (-128 to 127)
`short`	2	-2^{15} to $2^{15} - 1$ (-32768 to 32767)
`int`	4	-2^{31} to $2^{31} - 1$ ($\approx \pm 2 \times 10^9$)
`long`	8	-2^{63} to $2^{63} - 1$ ($\approx \pm 10^{19}$)
`float`	4	$\approx \pm 3 \times 10^{38}$ (limited precision)
`double`	8	$\approx \pm 10^{308}$ (limited precision)

Primitive Data Types

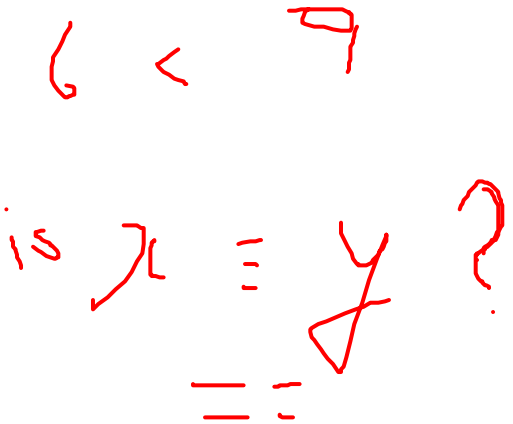
- Each type has certain *operations* that apply to it
- Common operations for primitive number types: + - * / (division) % (modulus / remainder)
  - Type of result is same as type of operands

- Use operations to construct **expressions**, which have values that can be assigned or used as operands
  - E.g., answer = (2 + 4) * 7;
  - count = count + 1;

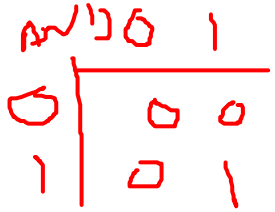
Operations for Number Types

Operations for Number Types (cont)

- Comparison operations also work for number type
  - < : less than
  - <= : less than or equal to
  - > : greater than
  - >= : greater than or equal to
  - == : equal to
  - != : not equal to
- Comparisons return boolean values
  - E.g., 5 != 4 returns true



- **&& (AND)** is true if both operands are true
  - E.g., int x = 5; then (x != 4) && (x > 3) is true
- **|| (OR)** is true if either operand is true
  - E.g., int x = 5; then (x != 4) || (x < 3) is true
- Both are short-circuit operations: they only evaluate the second operand if necessary
  - E.g., int x = 5; then (x != 4) || expr is true no matter what expr is because x != 4 is true
- **! (NOT)** is true if its operand is false
  - E.g., int x = 5; !(x == 4) is true



Operations for booleans

- **++x** is a special expression that increments x (for any variable x) and returns the incremented value
  - E.g., if x is 7, ++x is 8, and after that, x is 8
  - Called "pre-increment" because it increments variables *before* returning
- **--x** (pre-decrement) is similar: it decrements x and returns it

- **x++** (post-increment) returns x and then increments it
  - E.g., if x is 7, x++ is 7, and after that, x is 8
- **x--** (post-decrement) returns x and then decrements it

Pre / Post Increment / Decrement

Pre/Post Increment/Decrement

What will this code print?

```
int x = 10; int y = 5;
System.out.println(x++ - ++y);
```

10 − 6

A. 3
B. 4
C. 5
D. 6
E. 7

Quick Poll

Pre/Post Increment/Decrement

What will this code print?

```
int x = 10; int y = 5;
System.out.println(x++ - ++y);
```

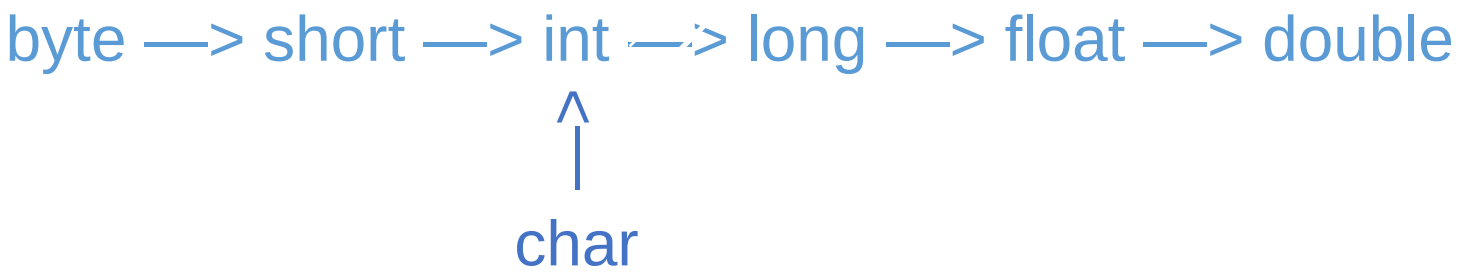
A. 3
B. 4
C. 5
D. 6
E. 7

Quick Poll

- Pre/post increment/decrement can be confusing (like the last example)
- They can also be used as statements rather than expressions
  - E.g., ++x; or x++;
  - Used as statements, these both just increment x

Pre/Post Increment/Decrement Use

- Primitive operations work on operands of the *same* type
- But Java can convert types automatically
- A *widening* conversion converts a number to a wider type (so the value can always be converted successfully)
- Automatic conversions in Java



Type Conversions

- *Narrowing* conversions are also possible. But they must be performed explicitly using a ***cast***
- Cast is specified by writing the name of the type to convert to in parentheses before the value to be converted
  - E.g., short x; int y = 50; x = (short) y;
- Cast can also be used to explicitly ask for a widening conversion

```
int sum;
int count;
//compute sum and count …
double average = (double) sum / count;
```

Casting

- Precedence of 2 operators, say $\odot$ and $\oplus$ determines whether $a \odot b \oplus c$ is read as:
  - $(a \odot b) \oplus c$ ($\odot$ has higher precedence), or
  - $a \odot (b \oplus c)$ ($\odot$ has lower precedence)
  - *E.g.*, 2+3*4 (* has higher precedence)
- Associativity determines whether $a \odot b \odot c$ is read as:
  - $(a \odot b) \odot c$ (left associativity), or
  - $a \odot (b \odot c)$ (right associativity)
  - *E.g.*, 3−2−1 (− associates left)
- Java's rules are mostly as you would expect
- When it doubt, just put in parentheses

Precedence and Associativity

Symbol	Associativity
. (method invocation)	
++ --	
- (unary negation)	
(*type*) casts	
* / %	Left
+ -	Left
< > <= >=	Left
== !=	Left
&&	Left
||	Left
= += *= ...	Right

Operators, High to Low Precedence

- After running the following piece of code, what will be the value of x, y, and z?

```
int x = 10, y = 5;
int z;
z = --x - y * 5 + x * (y++ - 4);
```

A. x = 10, y = 5, z = 5

B. x = 9, y = 5, z = -7

C. x = 9, y = 5, z = 5

D. x = 9, y = 6, z = -7

Quick Poll

- After running the following piece of code, what will be the value of x, y, and z?

```
int x = 10, y = 5;
int z;
z = --x - y * 5 + x * (y++ - 4);
```

A. x = 10, y = 5, z = 5

B. x = 9, y = 5, z = -7

C. x = 9, y = 5, z = 5

D. x = 9, y = 6, z = -7

Quick Poll

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Outline

- **String** is a class type, not a *primitive* type, so strings are *objects*
- Specify a string constant by enclosing in double-quotes (“”)
  - E.g., String s = “Hello, World!”;

- Use backslash (\) to include double-quote and other special characters (e.g., % and \) in a string
  - E.g., println(“He said \”backslash (\\) is special!\””)
  - Prints “He said “backslash (\) is special!””
- Certain letters after backslash are treated specially
  - Eg., \n - new line, \t - tab character

Strings

- You can use + to append two strings
  - E.g., System.out.println("Hello " + "World");
  - Prints "Hello World"
- If either operand is string, + operation will turn the other into a string
  - E.g., System.out.println("a = " + a);
  - If a is 1, this prints "a = 1"
- Beware:
  - System.out.println("a + a = " + a + a);
  - Actually prints "a + a = 11", not "1 + 1 = 2"

corrected

String Operations

- String class has many more operations, e.g.:
- Assume String s, s2; int i, j;
  - s.length() returns length of the string
  - s.toUpperCase() returns ALL UPPER CASE version of the string
  - s.substring(i, j) returns the substring of s from character I through j-1, counting the first char at 0
  - s.equals(s2) returns true if s and s2 are identical
  - s.indexOf(s2) returns the first position of s2 in s
- Don't use ==, <, >= etc. to compare strings
- See String class in documentation for more operations
  - Java API 8: https://docs.oracle.com/javase/8/docs/api/

More String Operations

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Outline

- **printf** is like **print**, but it lets you control how data is formatted
- Method form:
  - System.out.printf(format-string, args …);
- Example:
  - System.out.printf("Average: %5.2f", average);
- *Format-string* is an ordinary string, but can contain *format specifiers*, one for each of the arguments (args)
  - A format specifier begins with **%**
  - It may have a number specifying how to format the next value in the args list
  - It ends with a letter specifying the type of the value
- Ordinary text in format-string is printed as is

Formatted Output

%X.Y

- The (optional) number(s) following % is/are interpreted:
  - **X** (before decimal point) specifies the *minimum* number of characters to be printed
    - The ***full*** number will be printed, even if it takes more characters than X
    - If X is omitted, the value will be printed in its minimum width
    - If X is negative, the value will be left-justified, otherwise right-justified
  - **Y** (after decimal point) specifies the number of digits of the value to print after the decimal point
    - If Y is omitted, Java decides how to format

Format specifiers

The final letter in a format specifier can be:

`d`	format an integer (no fractional part)
`s`	format a string (no fractional part)
`c`	format a character (no fractional part)
`f`	format a float or double
`e`	format a float or double in exponential notation
`g`	like either `%f` or `%e`, Java chooses
`%`	output a percent sign (no argument)
`n`	end the line (no argument)

- Good format for money: `$%.2f`

Format Letters

```
public class PrintExample
{
    public static void main(String [] args)
    {
        String s = "string";
        double pi = 3.1415926535;
        System.out.printf("\"%s\" has %d characters %n", s, s.length());
        System.out.printf("pi to 4 places: %.4f%n", pi);
        System.out.printf("Right>>%9.4f<<", pi);
        System.out.printf(" Left >>%-9.4f<<%n", pi);
    }
}
```

Generated Output

```
"string" has 6 characters
Pi to 4 places: 3.1416
Right>>   3.1416<< Left>>3.1416   <<
```

Formatted Output Example

How many characters appear before the decimal point in a number `x` printed with `printf("%6.2f", x)`?

A. I don't know
B. 2
C. 3
D. 4
E. 6

Quick Poll

How many characters appear before the decimal point in a number x printed with `printf("%6.2f", x)`?

A. I don't know
B. 2
C. 3
D. 4
E. 6

Quick Poll

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Outline

- When your program is run, it can be given arguments on the command line
  - E.g., javac Hello.java ("Hello.java" is an argument for javac)
- Allows the user to give information to the program
- For the boilerplate we've been using, the command line arguments can be referred to as
  - First argument: args[0]
  - Second argument: args[1], etc..
- Each of these arguments is a string

Handling Command Line Inputs

```
// print out a friendly greeting
public class Hello2 {
    public static void main(String[] args) {
        System.out.println("Hello, " + args[0] + "!");
    }
}
```

Program Use

```
frege% java Hello2 Peter
Hello, Peter!
frege% java Hello2
Exception in thread "main" java.lang.ArrayIndexOutOfBo
undsException: 0
        at Hello2.main(Hello2.java:4)
```

Command Line Input Example

- To converts string to int:
  `Integer.parseInt(string)`

```
// Print double the command line integer
public class Hello3 {
    public static void main(String[] args) {
        System.out.println("Twice your number is "
                           + 2 * Integer.parseInt(args[0]));
    }
}
```

Program Use

```
 frege% java Hello3 4
Twice your number is 8
```

Handling Command Line Inputs

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Outline

Reading Console Input

- Interactive programs get input while running
- Java 5 introduces the `Scanner` class for this
- To use `Scanner`:
  - Add this near top of source file:

```
import java.util.Scanner;
```

  - Create scanner: add this in `main` before reading input:

```
Scanner keyboard = new Scanner(System.in);
```

  - Use `keyboard` as needed to read input
  - *E.g.*, this reads (rest of) current line as a string:

```
String line = keyboard.nextLine();
```

- Other methods to read from a `Scanner` variable `keyboard`:

What	Type	Expression
One word	`String`	`keyboard.next()`
One integer	`int`	`keyboard.nextInt()`
One double	`double`	`keyboard.nextDouble()`

- Similar methods to read other types; see documentation
- These all skip over whitespace and read one "word"
- Whitespace includes spaces, tabs, and newlines
- Error if text is not of expected type

Reading Console Input

```
import java.util.Scanner;
public class ScannerExample {
    public static void main(String[] args) {
        int num1 = Integer.parseInt(args[0]);
        Scanner kbd = new Scanner(System.in);
        System.out.print("Enter second number: ");
        int num2 = kbd.nextInt();
        System.out.println(num1 + " * " + num2 +
                           " = " + num1*num2);
    }
}
```

```
frege% java ScannerExample 6
Enter second number: 7
6 * 7 = 42
```

Command Line and Scanner Example

- `nextLine()` reads up to and including newline
- Others do not read after the next word
- After `next`, `nextInt`, or `nextDouble`, `nextLine` just reads rest of current line (maybe nothing!)
- To read a number on one line followed by the next whole line:

```
int num = keyboard.nextInt();
keyboard.nextLine(); // throw away rest of line
String line = keyboard.nextLine();
```

- Ideally, avoid mixing `nextLine` with the others

Pitfall: Mixing with nextLine

```
Scanner kbd = new Scanner(System.in);
int n       = kbd.nextInt();
String s1   = kbd.nextLine();
String s2   = kbd.nextLine();
```

Console input (on 3 lines):

```
1
+ 2
= 3
```

A. `s1 = "+ 2", s2 = "= 3"`
B. `s1 = "", s2 = "+ 2"`
C. `s1 = "= 3", s2 = ""`

Quick Poll: What are s1 and s2 after

```
Scanner kbd = new Scanner(System.in);
int n       = kbd.nextInt();
String s1   = kbd.nextLine();
String s2   = kbd.nextLine();
```

Console input (on 3 lines):

```
1
+ 2
= 3
```

A. s1 = "+ 2", s2 = "= 3"

B. s1 = "", s2 = "+ 2"

C. s1 = "= 3", s2 = ""

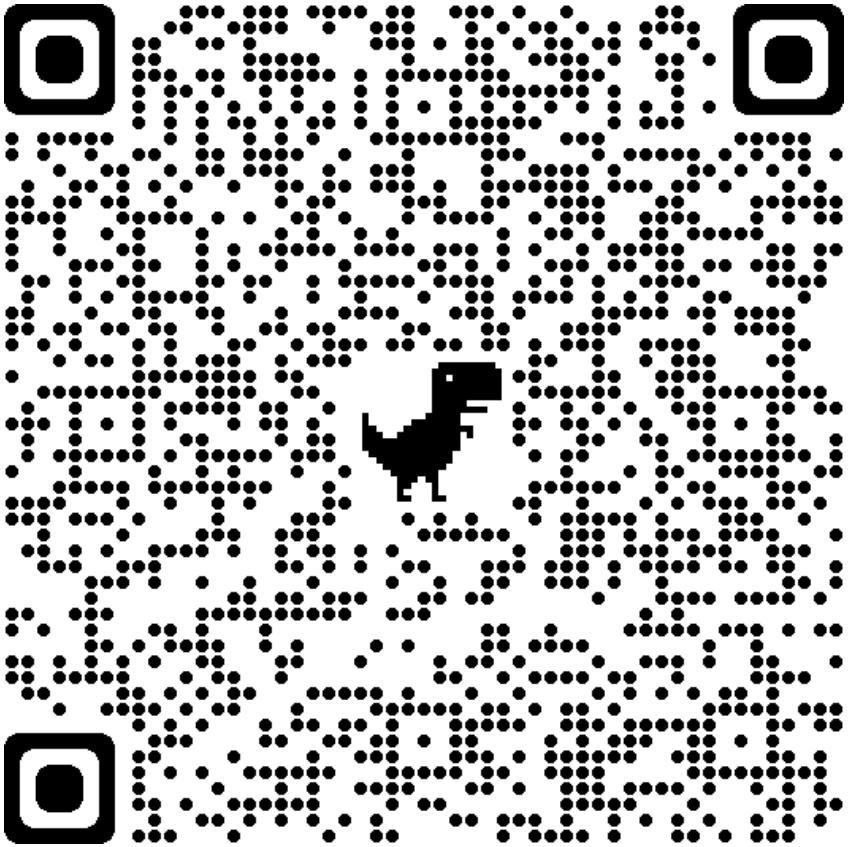
Quick Poll: What are s1 and s2 after

- Operations for primitive data types & type conversions
  - How to use different operators
  - How to identify/specify the precedence of the operators in an expression/a statement
  - How to convert between data types
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

Summary

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections



http://go.unimelb.edu.au/5o8i.

Programming and Software Development
COMP90041

Lecture 3

Flow Control

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 20-02-2021

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

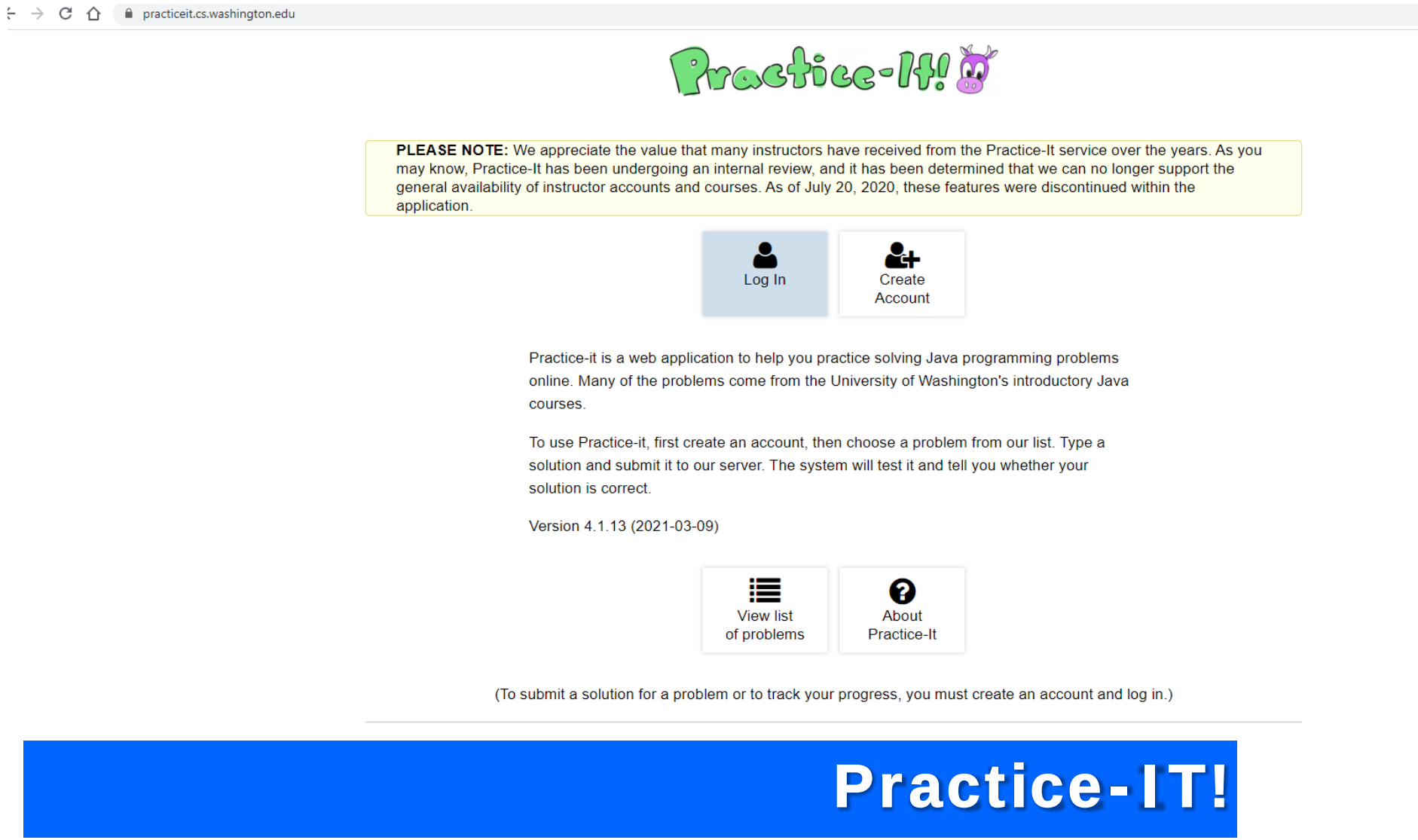
Review

- I hope all of you have now installed an IDE and practice simple java programs.
- Some of you might have missed a tutorial on Week 2 due to any issues, I have uploaded a video of an example tutorial for Week 2 for your benefit. If you have issues with IDE installation, please go through this. Note that this arrangement **is only for Week 2**. Due to privacy issues, we cannot record tutorials.
- Please go prepared to the workshop by trying to solve yourselves the questions.
- Remember, tutorials are **not** recorded, there is a high correlation of success in the subject to the regular attendance of tutorials.

Agenda

The Agenda for this week:

- Topics: Chapter 3 of the textbook
  - Branching mechanisms
  - Evaluating Boolean expressions
  - Different ways of constructing Loops
  - Debugging
- Tutorial – Week 3
  - Practice small Java programs
  - Formatted output
  - Running command line arguments



Practice-IT!

- Java's control statements allow you to control execution of code
- Conditional statements determine which statements to execute, possibly bypassing some
- Loop statements repeat some statements some number of times, under programmer control
- Programmer writes the program; user runs it
- It's up to the programmer to control the program based on the situation, including user actions

Flow Control

- **if** statement decides whether or not to execute a statement based on a **boolean** expression
- Form:
  - **if (expr) Statement**
- Executes the **Statement** if the **expr** is true, otherwise it does nothing
- The parenthesis are required
- The **expr** must be Boolean
  - Use ! = 0 to test an int
- E.g. negate x if It's negative

```
if (x < 0) x = -x;
```

- Most often, you need to execute multiple statements if the condition is true
- A **compound statement** turns multiple statements into a single statement that can be used in an **if**
- Also used in the other constructs in this lecture Form:
- **{ Statement1; · · · .... Statementn; }**
- Don't follow the brace with semicolon
- This is a single statement that executes

Statement1; · · · Statement n; in turn

```
if (x < 0) {
    System.out.println(x + " is negative!");
    x = -x;
}
```

Compound Statements

- What's wrong with this?

```
if (x < 0)
System.out.println(x + " is negative!");
x = -x;
```

- Best practice: always use braces, even for only one
- Statement:

```
if (x < 0){
 x = -x;
}
```

- Possible exception: whole if statement on one line
  - Unlikely to try to t another statement on the same line

```
if (x < 0) x = -x;
```

Best Practice

- Form:

- if (expr) **Statement1** else **Statement2**

- Executes **Statement1** if the expr is true, else executes **Statement2**

- Always executes exactly one of the statements
- Also best practice to surround **Statement1** and **Statement2** with braces

If-Else

- Always use indentation to show code structure
  - More indented code is part of less indented code
  - Indent one level per nesting level of braces
  - Not required by Java, but demanded by human readers
- One common layout:

```
if (x < 0)
{
    System.out.println("negative");
}
else
{
    System.out.println("non-negative");
}
```

Code Layout

- A more compact layout:

```
if (x < 0) {
System.out.println("negative");
} else {
System.out.println("non-negative");
}
```

- Amount to indent for each level:
  - 1 is too little; more than 8 too much
  - 4 is popular
- Beware of tabs: they mean different levels of indentation to different programs
  - 8 columns is standard
  - Best to avoid tabs altogether

Code Layout

- Java has no special form for handling a chain of conditions
- Just nest one **if-else** in the **else** part of another

```
if (x < 0) {
System.out.println("negative");
 } else if (x == 0) {
System.out.println("zero");
} else {
System.out.println("positive");
}
```

- Nest **if** and **if-else** within one another to any
- depth
  - Braces also makes this easier to read

Else If

- Java also has an if-else expression:

```
expr1 ? expr2 : expr3
```

> If **expr1** is **true** value is **expr2**
> If **expr1** is **false** value is **expr3**

- This:

- ```
  lesser = x < y ? x : y;
  ```
  does exactly the same as this:

```
if (x < y) {
    lesser = x;
} else {
lesser = y;
}
```

"Ternary Operator"
```

- **switch** statement chooses one of several cases
- based on an **int**, **short**, **byte**, or **char** value
- As of Java 7, it can also be a **String**: more useful
- Form:
- switch (**expr**) {

```
        case value1 :
          statements...
         break;
        .
        .
case valuen :
        statements...
        break;
    }
```

Switch

- Execution begins by evaluating the expression
- It then looks for a **case** with matching **value**
- If it finds one, it begins executing with the next
- statement
- It stops executing when it reaches a **break** or the
- end of the **switch**
- Cases can be put in any order

Switch

- As a special case, can use **default** in place of one **case *value***
- If no **case *value*** matches, the code after the **default**: is executed, up to the next break;

- If no **case *value*** matches and there is no **default**:, **switch** statement finishes without executing any of the statements

Default

- If there is no break before the next case label,
- Java keeps executing until the next break
- Very easy to forget a break
- Best practice: even put break at end of last case
  - You may later add a new case after the last one
- If you leave out a break on purpose, put in a comment saying why
  - So whoever reads code (including you, later) knows it was omitted on purpose
- Exception: same code for multiple cases: just put common case labels together, followed by code

Pitfall: Missing break

```
switch (ch) {
case '.':
    System.out.print("dot ");
    break;
case '-':
case '_':
    System.out.print("dash ");
    break;
case ' ':
    System.out.println(); // start new line
    break;
default:
    System.out.println("\nbad character '" + ch + "'")
    break;
}
```

Example: Spell Out Morse Code

- Form:
- **while (*expr*) Statement**
- If ***expr*** is ***true*** then:
  - Execute the ***Statement*** , then
  - Then go back and check ***expr*** again
  - Keep executing ***Statement*** as long as expr is true
- Stops when ***expr*** is **false** at top of loop
- Use to execute ***Statement*** an unlimited number of times, as long as ***expr*** is true
- Only useful if ***Statement*** can change value of ***expr***
- Best practice again: put ***Statement*** in braces unless whole while fits on one line

While

```
public class whileExample {
    public static void main(String[] args) {
        int i = 1;
        int limit = 10;
        int sum = 0;
        while (i <= limit) {
            sum += i;
            ++i;
        }
        System.out.println("The sum is " + sum);
    }
}
```

- Generated Output

  The sum is 55

While Example

- Form:
- **do *Statement* while (*expr*)**
- First execute *Statement*
- Then, if *expr* is `true`, go back and do it again
  - Keep executing *Statement* as long as *expr* is true
- Stops when *expr* is `false` at bottom of loop Use when you must execute *Statement* before testing *expr*
- Only useful if *Statement* can change value of *expr* Best practice again: put *Statement* in braces unless whole `while` fits on one line

Do While

```
public class dowhileExample {
    public static void main(String[] args) {
        int i = 1;
        int limit = 10;
        int sum = 0;
        do {
            sum += i;
            ++i;
        } while (i <= limit);
        System.out.println("The sum is " + sum);
    }
}
```

- Generated Output:
- The sum is 55

do while Example

- while executes *Statement* zero or more times
- do while executes *Statement* one or more times
- Use while if you need to check a condition every time before executing the *Statement*
- Use do while if you need to execute the *Statement* before evaluating the *expr* every time
- Changing limit to 0 in the while example will
- print a sum of 0
- Changing limit to 0 in the do while example will print a sum of 1! That's wrong!
- while is more commonly used

So What's The Difference?

- `for` is like `while` with initialization and increment Form:

- **`for (init ; test ; update ) Statement`**

- *init* is for variable initialisations, *e.g.*, x = 0 *test* is a boolean expression to decide whether to execute Statement
- *update* is executed after each iteration
- Useful to execute a specific number of iterations Equivalent to:

- ```
  init ;
  while(test) {
        Statement;
        update;
  }
  ```

For
  ```

```
public class forExample {
    public static void main(String[] args) {
        int limit = 10;
        int sum = 0;
        for (int i = 0; i <= limit; ++i) {
            sum += i;
        }
        System.out.println("The sum is " + sum);
    }
}
```

- Generated Output:
- The sum is 55

for Example

- Any of *init* , *test* and *update* parts can be omitted
  - Infinite loop if *test* is omitted, but see below
- Variables declared in *init* part are scoped to the for: not available after the loop
- But you can declare variable before loop, and just initialise it in the *init* part
- Can include multiple initializations and updates by separating them with commas
- But if you put a declaration in the *init* part, you can only specify one type (not so useful)
- Only one *test* part is allowed, but can use && and || to define it

For

- Inside a `for`, `while` or `do while` loop, a `break` terminates the (innermost) loop immediately
- This is useful inside an `if` inside a loop

- A `continue` statement immediately returns to the top of the innermost loop and continues from there Can immediately exit whole program with

`System.exit(0);` statement

- Use 0 to indicate "success" and > 0 to indicate error Will see a better way to handle errors later. . .

break and continue

- Infinite loop: loop never terminates
  - Forget to update the counter
  - Use wrong test
- Best practice: use < or <= (or > or >=) in loop test, not == or !=
- Off-by-one (fence post) error: one too many or few iterations
  - Start or end too low or too high
  - Use < instead of <= or vice-versa
- For n iterations, do one of:

```
for (i=0 ; i<n ; ++i) or
for (i=1 ; i<=n ; ++i)
```

Pitfall: Common Loop Errors

- Use `assert(`*test*`)` to sanity-check your code Often program errors go undetected for a long time Very difficult to trace symptom back to cause
- Worst thing a program can do is not crash, but run normally producing wrong results
- `assert` stops the program if something is wrong *E.g.*, if at some point `x` must always be positive, add this statement at that point:

```
assert x > 0;
```

- Assertions not normally checked
  - Turn on checking by running program with:

```
java -enableassertions ProgramName
```

assert

- Use `if` or `if else` or `switch` to conditionally execute a statement
- Enclose multiple statements in {braces} to treat as a single statement
- Remember: end each `switch` case with a `break` Use `while` or `do while` or `for` loops to repeat a statement
- Use `break` to terminate loop immediately
- Use `contine` to restart a loop immediately

Summary

Programming and Software Development
COMP90041

Lecture 4

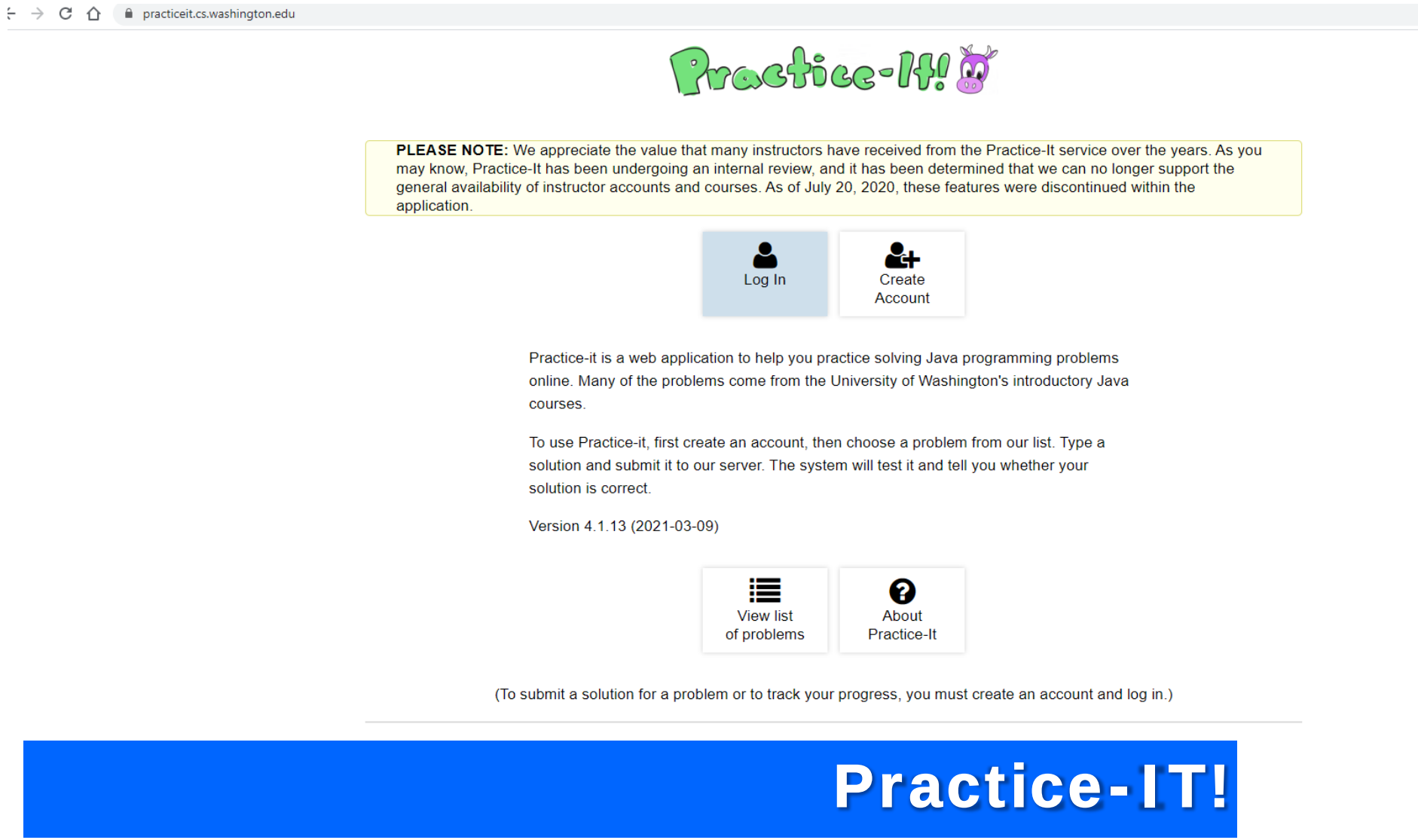
Classes and Methods

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 20-02-2021

- Topics: Chapter 3 of the textbook
  - Branching mechanisms
  - Evaluating Boolean expressions
  - Different ways of constructing Loops
  - Debugging
- Tutorial – Week 3
  - Practice small Java programs
  - Formatted output
  - Running command line arguments

Review



Practice-IT!

Write a program that prints out the following triangle to the screen using a nested for loop.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

```
1  //Similar solutions proposed by
2  //Jeffrey Lau and Edwin Sutanto
3  import java.util.Scanner;
4  public class QuickTest5
5  {
6    public static void main(String[] args) {
7      Scanner keyboard = new Scanner(System.in);
8      System.out.println("Number of rows:");
9      int rows = keyboard.nextInt();
10     String formatString = "%" + rows + "s%n";
11     String stars = "";
12     for (int i = 1; i <= rows; i++) {
13       stars = stars + "*";
14       System.out.printf(formatString, stars);
15     }
16   }
17 }
```

Quick Test - 5

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Outline

- A Java program is made up of interacting objects from various classes.
- A method is an operation defined by a class
- i.e., it defines how to do something
- The Java library defines many methods And you can define your own
- Similar to functions, subroutines, procedures in other languages
- Java supports two kinds of methods:
  - Class or `static` methods, and
  - Instance or `non-static` methods
- Instance methods are more common, but Class methods are simpler, so we start there

Methods and Classes

- **Class definitions**
  - **Class structure**
  - **Variables**
  - **Methods**
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Outline

- Classes are central to Java
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes

A Java program consists of objects from various classes Interacting with one another

Introduction

- A class is a type and you can declare variables of a class type (e.g., Car myCar)
- A value of a class type is called an ***object*** or *an instance of the class*
- An object has both ***data*** and ***actions***
  - **actions** are called **methods**
- Each object can have different data, but all objects of a class have the same types of data and all objects of a class have the same methods (e.g., myCar vs yourCar)

Terminology

- A primitive type value is a single piece of data
- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value

Primitive Type Values vs. Class Type Values

- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition

Class Definition

```
public class Class_Name
{
  Instance_Variable_Declartion_1
  Instance_Variable_Declartion_2
  ...
  Instance_Variable_Declartion_Last

  Method_Definition_1
  Method_Definition_2
  ...
  Method_Definition_Last
}
```

Data

Actions

Java Class Structure

```
public class Class_Name
{
  Instance_Variable_Declartion_1
  Instance_Variable_Declartion_2
  ...
  Instance_Variable_Declartion_Last

  Method_Definition_1
  Method_Definition_2
  ...
  Method_Definition_Last
}
```

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        ...
    }
}
```

The simple HelloWorld program follows the Java class structure

Java Class Structure

- An object of a class is named or declared by a variable of the class type:
  ClassName classVar;
- The **new** operator must then be used to create the object and associate it with its variable name:
  classVar = new ClassName();
- These can be combined as follows:
  ClassName classVar = new ClassName();

E.g. Date.java and DateDemo.java

The new Operator

- Instance variables can be defined as in the following two examples
  - Note the **public** modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Instance Variables and Methods

- Method definitions are divided into two parts: a *heading* and a *method body:*

```
public void myMethod()          <-- Heading
{
   code  to perform some action  } Body
   and/or compute a value        }
}
```

- Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body

Instance Variables and Methods

- Reminder: a Java file must be given the same name as the class it contains with an added **.java** at the end
  - For example, a class named **MyClass** must be in a file named **MyClass.java**
- For now, your program and all the classes it uses should be in the same directory or folder

File Names and Locations

```java
public class Date
{
  public int day;
  public int month;
  public int year;

  public void writeOutput()
  {
    System.out.println(day + "/" +
                month + "/" + year);
  }
}
```

Date.java

Compilation:

javac DateDemo.java

Execution:

java DateDemo

```java
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date date1 = new Date();
    Date date2 = new Date();

    date1.day = 31; //data initialization/update
    date1.month = 12;
    date1.year = 2012;
    System.out.println("date1:");
    date1.writeOutput(); //method invocation

    date2.day = 4;
    date2.month = 7;
    date2.year = 1776;
    System.out.println("date2:");
    date2.writeOutput();
  }
}
```

DateDemo.java

Example-1: Simple class definition

- There are two kinds of methods:
  - Methods that compute and return a value
  - Methods that perform an action
    - This type of method does not return a value, and is called a **void** method
- Each type of method differs slightly in how it is defined as well as how it is (usually) invoked

More About Methods

- A method that returns a value must specify the type of that value in its heading:

  public typeReturned methodName(parameter_List)

- A **void** method uses the keyword **void** in its heading to show that it does not return a value :

  public void methodName(parameter_List)

Method definitions

- A program in Java is just a class that has a **main** method
- When you give a command to run a Java program, the run-time system invokes the method **main**
- Note that **main** is a **void** method, as indicated by its heading:

```
public static void main(String[] args)
```

`main` is a `void` Method

- The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

```
public <void or typeReturned> myMethod()
{
   declarations (for local variables)
   statements
}
```

Method body

- The body of a method that returns a value must also contain one or more **return** statements
  - A **return** statement specifies the value returned and ends the method invocation:

    return Expression;
  - Expression can be any expression that evaluates to something of the type returned listed in the method heading

`return` Statements

- A **void** method need not contain a **return** statement, unless there is a situation that requires the method to end before all its code is executed
- In this context, since it does not return a value, a **return** statement is used without an expression:

```
return;
```

`return` Statements

- An invocation of a method that returns a value can be used as an expression anyplace that a value of the **typeReturned** can be used:

```
typeReturned tRVariable;
tRVariable = objectName.methodName();
```

- An invocation of a **void** method is simply a statement:

```
objectName.methodName();
```

Method invocation

- An invocation of a method that returns a value of type **`boolean`** returns either **`true`** or **`false`**
- Therefore, it is common practice to use an invocation of such a method to control statements and loops where a Boolean expression is expected
  - **if-else** statements, **while** loops, etc.

Methods That Return a Boolean Value

- A method that returns a value can also perform an action

- If you want the action performed, but do not need the returned value, you can invoke the method as if it were a **void** method, and the returned value will be discarded:

```
objectName.returnedValueMethod();
```

Any Method Can Be Used As a `void` Method

```java
public class Date
{
  public int day;
  public int month;
  public int year;
  //a void method
  public void writeOutput()
  {
    System.out.println(day + "/" +
                month + "/" + year);
  }
  //a method that returns a value
  public int getYear()
  {
    return year;
  }
}
```

Date.java

```java
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date date1 = new Date();

    date1.day = 31; //data initialization/update
    date1.month = 12;
    date1.year = 2012;
    System.out.println("date1:");
    date1.writeOutput(); //void method invocation

    int year = date1.getYear(); //method invocation
    System.out.printf("Year: %d\n", year);
  }
}
```

DateDemo.java

Example-2: Types of methods

- A variable declared within a method definition is called a ***local variable***
  - All method parameters are local variables
- If two methods each have a local variable of the same name, they are still two ***entirely different*** variables

- **Note:** Some programming languages include another kind of variable called a *global* variables. The Java language does **not** have global variables

Local Variables

- A *block* is another name for a compound statement, that is, a set of Java statements enclosed in braces, **{}**
- A variable declared within a block is local to that block
  - When the block ends, all variables declared within the block disappear
- **Note:** in Java, you cannot have two variables with the same name inside a single method definition (e.g., inside a block and outside a block)

Blocks

- You can declare one or more variables within the initialization portion of a **for** statement

```
int sum = 0;
for (int i = 1; i <= 100; i++)
{
   sum = sum + i;
}
```

- The variable **i** is local to the **for** loop, and cannot be used outside of the loop
- If you need to use such a variable outside of a loop, then declare it outside the loop

Declaring Variables in a for Statement

- The methods seen so far have had no parameters, indicated by an empty set of parentheses in the method heading
- Some methods need to receive additional data via a list of ***parameters*** in order to perform their work
  - These *parameters* are also called ***formal parameters***

Parameters of a Primitive Type

- A parameter list provides a description of the data required by a method
  - It indicates the number and types of data pieces needed, the order in which they must be given, and the local name for these pieces as used in the method

public double myMethod(int p1, int p2, double p3)

Parameters of a Primitive Type

- When a method is invoked, the appropriate values must be passed to the method in the form of ***arguments***
  - Arguments are also called ***actual parameters***
- The *number and order* of the arguments must exactly match that of the parameter list
- The type of each argument must be compatible with the type of the corresponding parameter

```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

Parameters of a Primitive Type

- If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion
  - In the preceding example, the **`int`** value of argument **`c`** would be cast to a **`double`**
  - A primitive argument can be automatically type cast from any of the following types, to any of the types that appear to its right:

`byte`→**`short`**→**`int`**→**`long`**→**`float`**→**`double`**

`char`

Parameters of a Primitive Type

- In the preceding example, the value of each argument (not the variable name) is plugged into the corresponding method parameter
  - This method of plugging in arguments for formal parameters is known as the

call-by-value mechanism

```
public double myMethod(int p1, int p2, double p3)


int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

Parameters of a Primitive Type

- A parameter is filled in by the value of its corresponding argument
- A parameter is actually a local variable
- When a method is invoked, the value of its argument is computed/evaluated, and the corresponding parameter (i.e., local variable) is initialized to this value
- Even if the value of a formal parameter is changed within a method (i.e., it is used as a local variable) the *value of the argument cannot be changed*

Call-by-value mechanism

```java
public class Date
{
  public int day;
  public int month;
  public int year;
  //a void method
  public void writeOutput()
  {
    System.out.println(day + "/" +
                month + "/" + year);
  }
  //a method that returns a value
  public int getYear()
  {
    return year;
  }
  //a method with parameters
  public void setDate(int aDay,
                int aMonth, int aYear)
  {
    day = aDay;
    month = aMonth;
    year = aYear;
  }
}
```

Date.java

```java
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date date1 = new Date();

    date1.setDate(31, 12, 2012);
    System.out.println("date1:");
    date1.writeOutput(); //void method invocation

    int year = date1.getYear(); //method invocation
    System.out.printf("Year: %d\n", year);
  }
}
```

DateDemo.java

Example-3: Primitive parameters

- Use a method parameter as a local variable
  - Update the value of the parameter inside the method

Another example

Display 4.6 A Formal Parameter Used as a Local Variable

This is the file `Bill.java`.

```
1   import java.util.Scanner;

2   public class Bill
3   {
4       public static double RATE = 150.00; //Dollars per quarter hour

5       private int hours;
6       private int minutes;
7       private double fee;
```

(continued)

A Formal Parameter Used as a Local Variable (Part 1 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

```
 8       public void inputTimeWorked()
 9       {
10           System.out.println("Enter number of full hours worked");
11           System.out.println("followed by number of minutes:");
12           Scanner keyboard = new Scanner(System.in);
13           hours = keyboard.nextInt();
14           minutes = keyboard.nextInt();
15       }

16       public double computeFee(int hoursWorked, int minutesWorked)
17       {
18           minutesWorked = hoursWorked*60 + minutesWorked;
19           int quarterHours = minutesWorked/15; //Any remaining fraction of a
20                                                // quarter hour is not charged for.
21           return quarterHours*RATE;
22       }

23       public void updateFee()
24       {
25           fee = computeFee(hours, minutes);
26       }
```

computeFee uses the parameter `minutesWorked` *as a local variable.*

Although `minutes` *is plugged in for* `minutesWorked` *and* `minutesWorked` *is changed, the value of* `minutes` *is not changed.*

(continued)

A Formal Parameter Used as a Local Variable (Part 2 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

```
27      public void outputBill()
28      {
29          System.out.println("Time worked: ");
30          System.out.println(hours + " hours and " + minutes + " minutes");
31          System.out.println("Rate: $" + RATE + " per quarter hour.");
32          System.out.println("Amount due: $" + fee);
33      }
34  }
```

(continued)

A Formal Parameter Used as a Local Variable (Part 3 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

This is the file `BillingDialog.java`.

```
1  public class BillingDialog
2  {
3     public static void main(String[] args)
4     {
5          System.out.println("Welcome to the law offices of");
6          System.out.println("Dewey, Cheatham, and Howe.");
7          Bill yourBill = new Bill();
8          yourBill.inputTimeWorked();
9          yourBill.updateFee();
10         yourBill.outputBill();
11         System.out.println("We have placed a lien on your house.");
12         System.out.println("It has been our pleasure to serve you.");
13       }
14 }
```

(continued)

A Formal Parameter Used as a Local Variable (Part 4 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

SAMPLE DIALOGUE

```
Welcome to the law offices of
Dewey, Cheatham, and Howe.
Enter number of full hours worked
followed by number of minutes:
3 48
Time worked:
2 hours and 48 minutes
Rate: $150.0 per quarter hour.
Amount due: $2250.0
We have placed a lien on your house.
It has been our pleasure to serve you.
```

A Formal Parameter Used as a Local Variable (Part 5 of 5)

- Do not be surprised to find that people often use the terms **parameter** and **argument** interchangeably
- When you see these terms, you may have to determine their exact meaning from context

Pitfall: Use of the Terms "Parameter" and "Argument"

- All instance variables are understood to have **`<the calling object>.`** in front of them
- If an explicit name for the calling object is needed, the keyword **`this`** can be used
  - **myInstanceVariable** always means and is always interchangeable with **this.myInstanceVariable**

The this Parameter

- **this *must*** be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
int someVariable = this.someVariable
```

The this Parameter

- What will happen if we make the following changes to the **setDate** method in Example-3?

```
public void setDate(int day, int month, int year)
{
    day = day;
    month = month;
    year = year;
}
```

Example-4: this Parameter

- The **`this`** parameter is a kind of hidden parameter
- Even though it does not appear on the parameter list of a method, it is still a parameter
- When a method is invoked, the calling object is automatically plugged in for **`this`**

The this Parameter

- Java expects certain methods, such as **`equals`** and **`toString`**, to be in all, or almost all, classes
- The purpose of **`equals`**, a **`boolean`** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use **`==`** to compare objects

```
public boolean equals(ClassName objectName)
```

- The purpose of the **`toString`** method is to return a **`String`** value that represents the data in the object

```
public String toString()
```

The methods equals and toString

```java
//equals and toString method
public boolean equals(Date otherDate)
{
  if ((otherDate.day == day)
     && (otherDate.month == month)
     && (otherDate.year == year))
    return true;
  else
    return false;
}
public String toString()
{
  return (day + "/" + month + "/" + year);
}
```

Newly added methods
Inside Date.java

```java
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date d1 = new Date();
    Date d2 = new Date();

    d1.setDate(31, 12, 2012);
    d2.setDate(31, 12, 2012);

    System.out.printf("%s and %s are %s\n",
      d1.toString(), d2.toString(),
      d1.equals(d2)?"the same":"not the same");
  }
}
```

DateDemo.java

Example-5: equals and toString methods

- Each method should be tested in a program
  - A program whose only purpose is to test a method is called a ***driver program***
- One method often invokes other methods, so one way to do this is to first test all the methods invoked by that method, and then test the method itself
  - This is called ***bottom-up testing***
- Sometimes it is necessary to test a method before another method it depends on is finished or tested
  - In this case, use a simplified version of the method, called a ***stub***, to return a value for testing

Testing Methods

- Class definitions
  - Class structure
  - Variables
  - Methods
- **Encapsulation**
  - **Access modifiers (e.g., public vs private)**
  - **Accessor and mutator methods**
- Overloading
- Constructors

Outline

- ***Information hiding*** is the practice of separating how to use a class from the details of its implementation
  - ***Abstraction*** is another term used to express the concept of discarding details in order to avoid information overload
- ***Encapsulation*** means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface

Information Hiding and Encapsulation

- The **API** or ***application programming interface*** for a class is a description of how to use the class
  - A programmer need only read the API in order to use a well designed class
- An **ADT** or ***abstract data type*** is a data type that is written using good information-hiding techniques

Important Acronyms: API and ADT

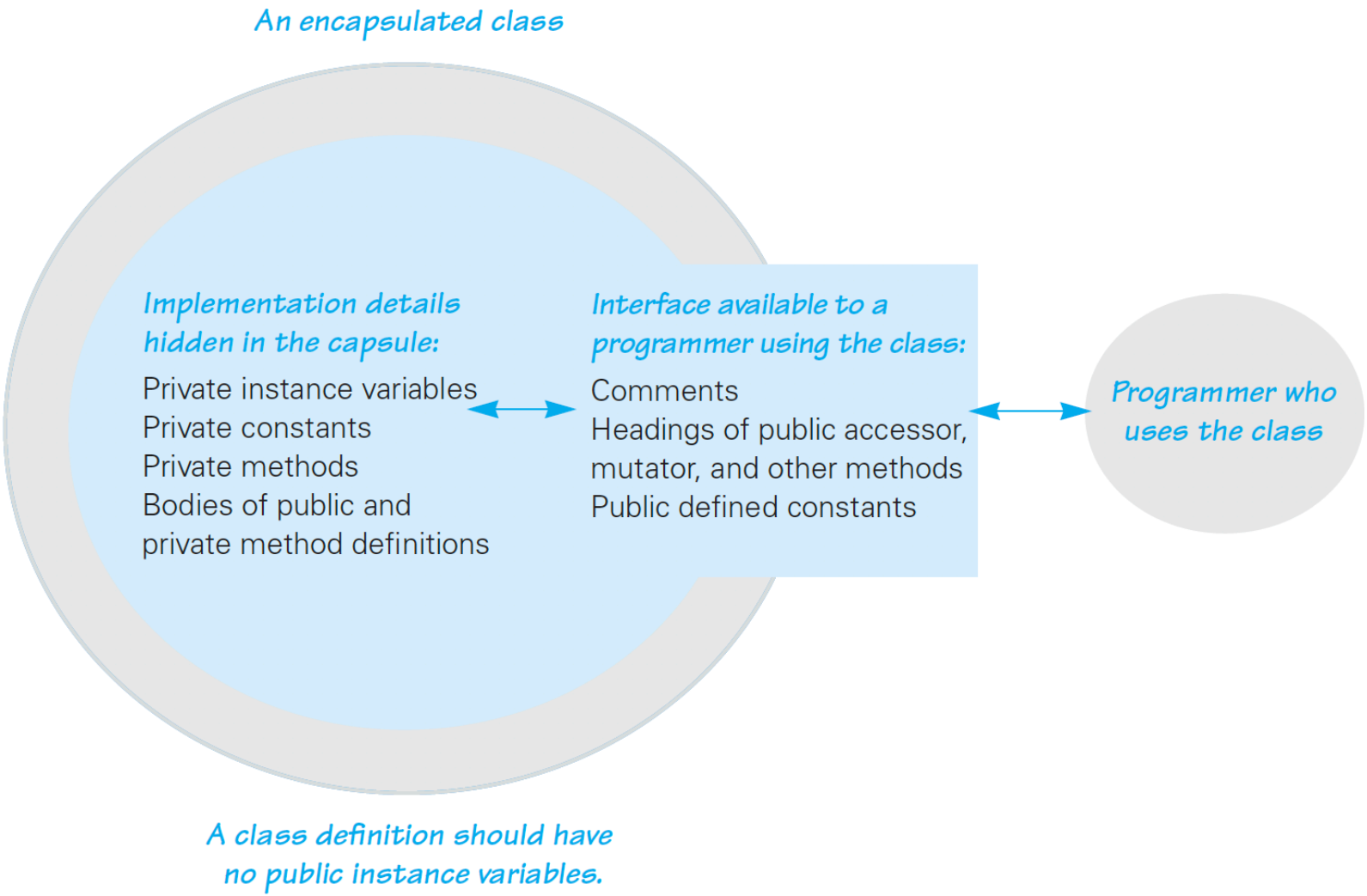
- The modifier **`public`** means that there are no restrictions on where an instance variable or method can be used
- The modifier **`private`** means that an instance variable or method cannot be accessed by name outside of the class
- It is considered good programming practice to make **all** instance variables **`private`**
- Most methods are **`public`**, and thus provide controlled access to the object
- Usually, methods are **`private`** only if used as helping methods for other methods in the class

`public` and **`private`** Modifiers

- ***Accessor*** methods allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **`get`**
- ***Mutator*** methods allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **`set`**

Accessor and Mutator Methods

Display 4.10 Encapsulation



Encapsulation

- Within the definition of a class, private members of **any** object of the class can be accessed, not just private members of the calling object

```
public boolean equals(Date otherDate)
{
  if ((otherDate.day == day)
     && (otherDate.month == month)
     && (otherDate.year == year))
    return true;
  else
    return false;
}
```

A Class Has Access to Private Members of All Objects of the Class

- The ***precondition*** of a method states what is assumed to be true when the method is called
- The ***postcondition*** of a method states what will be true after the method is executed, as long as the precondition holds
- It is a good practice to always think in terms of preconditions and postconditions when designing a method, and when writing the method comment

```
/**
Precondition: All instance variables of the calling object have
values.
Postcondition: The data in the calling object has been written to
the screen.
*/
public void writeOutput()
```

Preconditions and Postconditions

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- **Overloading**
- Constructors

Outline

- ***Overloading*** is when two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have different ***signatures***
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

Overloading

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- The interaction of overloading and automatic type conversion can have unintended results
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

Overloading and Automatic Type Conversion

- The signature of a method only includes the method name and its parameter types
  - The signature does **not** include the type returned
- Java does not permit methods with the same name and different return types in the same class

Pitfall: You Can Not Overload Based on the Type Returned

- Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
  - You may only use a method name and ordinary method syntax to carry out the operations you desire

You Can Not Overload Operators in Java

```java
public void setDate(int aDay,
            int aMonth, int aYear)
{
  day = aDay;
  month = aMonth;
  year = aYear;
}

public void setDate(int aDay,
            String aMonth, int aYear)
{
  day = aDay;
  month = convertMonth(aMonth);
  year = aYear;
}
```

Two setDate methods having different signatures in Date.java

Example-7

```java
//helper methods
private int convertMonth(String aMonth)
{
  if (aMonth.equalsIgnoreCase("January"))
    return 1;
  else if (aMonth.equalsIgnoreCase("February"))
    return 2;
  else if (aMonth.equalsIgnoreCase("March"))
    return 3;
  else if (aMonth.equalsIgnoreCase("April"))
    return 4;
  else if (aMonth.equalsIgnoreCase("May"))
    return 5;
  else if (aMonth.equalsIgnoreCase("June"))
    return 6;
  else if (aMonth.equalsIgnoreCase("July"))
    return 7;
  else if (aMonth.equalsIgnoreCase("August"))
    return 8;
  else if (aMonth.equalsIgnoreCase("September"))
    return 9;
  else if (aMonth.equalsIgnoreCase("October"))
    return 10;
  else if (aMonth.equalsIgnoreCase("November"))
    return 11;
  else if (aMonth.equalsIgnoreCase("December"))
    return 12;
  else
  {
    System.out.println("Fatal Error");
    System.exit(0);
    return 0; //Needed to keep the compiler happy
  }
}
```

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- **Constructors**

Outline

- A ***constructor*** is a special kind of method that is designed to initialize the instance variables for an object:
  public ClassName(anyParameters){code}
  - A constructor must have the same name as the class
  - A constructor has no type returned, not even **`void`**
  - Constructors are typically overloaded

Constructors

- A constructor is called when an object of the class is created using **new**
  ClassName objectName = new ClassName(anyArgs);
  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **new** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- If a constructor is invoked again (using **new**), the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use mutator methods instead

Constructors

- The first action taken by a constructor is to create an object with instance variables
- Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another

You Can Invoke Another Method in a Constructor

- Like any ordinary method, every constructor has a **`this`** parameter
- The **`this`** parameter can be used explicitly, but is more often understood to be there than written down
- The first action taken by a constructor is to automatically create an object with instance variables
- Then within the definition of a constructor, the **`this`** parameter refers to the object created by the constructor

A Constructor Has a `this` Parameter

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- If you include even one constructor in your class, Java will not provide this default constructor
- If you include any constructors in your class, normally you should provide your own no-argument constructor.

Include a No-Argument Constructor

- Instance variables are automatically initialized in Java
  - **boolean** types are initialized to false
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to **`null`**
- However, it is a better practice to explicitly initialize instance variables in a constructor
- **Note**: Local variables are not automatically initialized

Default Variable Initializations

```
//constructors
public Date()
{
  day = 1;
  month = 1;
  year = 1000;
}

public Date(int aDay, int aMonth, int aYear)
{
  day = aDay;
  month = aMonth;
  year = aYear;
}

public Date(int aDay, String aMonth, int aYear)
{
  day = aDay;
  month = convertMonth(aMonth);
  year = aYear;
}
```

Example-8: Constructors

- Class structure
- Instance variables and methods
- Different types of methods and their invocation
- Information hiding & Encapsulation
- Overloading methods
- Class constructors

Learning Outcomes