COMP2511

Week 2

THURSDAY 9AM - 12PM (H09A)
FRIDAY 10AM - 1PM (F10A)

Attendance

Attendance

If you for some reason are not able to attend the tutorial, please send me an email with a short description why to get the average participation mark for the week

VSCode Config Tips

Ensure that you open the correct folder

example: cs2511-project

Good

Bad

VSCode Config Tips

Ensure you have the correct Java extension installed

VSCode Config Tips

Ensure that you are running code using the "Run" button

```
You, last week | 1 author (You)
1  package example;
2
3  import java.util.Arrays;
4  import java.util.Scanner;
5
You, last week | 1 author (You)
6  /**
7   * Write a program that uses the `Scanner` class
8   * which reads in a line of numbers separated by spaces,
9   * and sums them.
10  */
11      You, last week • Week 1 …
12 public class Sum {
     Run | Debug
13     public static void main(String[] args) {
14         /**
15          * new - Creates a new Scanner object. (Think of it like C Malloc, but Java's
16          * garbage collection frees it)
17          * Scanner is an object that allows us to specify an input
18          * System.in == stdin in C
19          */
20         Scanner scanner = new Scanner(System.in);
21
```

GitLab Tips

How to view pipeline output

GitLab Tips

How to view pipeline output

GitLab Tips

How to view pipeline output

```
Running with gitlab-runner 13.8.0 (775dd39d)
  on COMP2511 Primary Runner frcgae5g
Preparing the "docker" executor                                                    00:04
Using Docker executor with image sneakypatriki/cs2511-basic:latest ...
Pulling docker image sneakypatriki/cs2511-basic:latest ...
Using docker image sha256:6b096604fbdcff132bec42cd058ffba4d9c817b76ebec2bc5597cc8e412fe063 for sneakypatriki/cs2511-basic:latest with digest sneakypatriki/cs2511-basic@sha256:73409fb773ed594c21365f434cb2d887459fd714c04beef73e90f7369783bc57 ...
Preparing environment                                                              00:01
Running on runner-frcgae5g-project-261312-concurrent-0 via cashewbread...
Getting source from Git repository                                                 00:01
Fetching changes with git depth set to 50...
Reinitialized existing Git repository in /builds/COMP2511/22T2/STAFF/repos/lab01/.git/
Checking out 794e0839 as master...
Removing src/average/Average.class
Skipping Git submodules setup
Executing "step_script" stage of the job script                                    00:02
$ bash tests.sh
--- /dev/fd/63
+++ /dev/fd/62
@@ -0,0 +1 @@
+The average is 3.5
Cleaning up file based variables                                                   00:00
ERROR: Job failed: exit code 1
```

Test File

- Compares the difference between your output and what is expected
- Runs test in order. Average -> splitter -> satellite
- No output == good

```
#!/bin/bash
function run_junit() {
    exercise=$1
    rm -rf bin/$exercise
    javac -d bin -cp "$JUNIT" $(find src/$exercise -name "*.java")
    java -jar "$JUNIT" -cp bin:src/$exercise --scan-class-path --disable-
}

cd src
# Average
javac average/Average.java || exit 1
diff <(java average/Average) <(echo "The average is 3.5") || exit 1

# Splitter
javac splitter/Splitter.java || exit 1
diff <(java splitter/Splitter <<< "Welcome to my humble abode") <(printf

# Satellite
javac satellite/Satellite.java || exit 1
diff <(java satellite/Satellite) <(printf "I am Satellite A at position 1

cd ..

JUNIT="lib/junit-platform-console-standalone-1.7.0-M1.jar"
```

Test Output

```
Executing "step_script" stage of the job script
$ bash tests.sh
--- /dev/fd/63
+++ /dev/fd/62
@@ -1 +1 @@
-The average is 3.0
+The average is 3.5
```

- The '-' (minus) line is your output
- The '+' (plus) line is expected

Code Review

- What is the difference between **super** and **this**?
- What about **super(...)** and **this(...)**?

```java
package shapes;

public class Shape {
    public String color;

    public Shape(String color) {
        System.out.println("Inside Shape constructor");
        this.color = color;
    }
}
```

```java
public class Rectangle extends Shape {
    public int height;
    public int width;

    public Rectangle(String color) {
        super(color);
        System.out.println("Inside Rectangle constructor with one argument");
    }

    public Rectangle(String name, int width, int height) {
        this(name);
        this.width = width;
        this.height = height;
        System.out.println("Inside Rectangle constructor with three arguments");
    }

    public static void main(String[] args) {
        Rectangle r = new Rectangle("red", 10, 20);
        System.out.println(r.color);
        System.out.println(r.width);
        System.out.println(r.height);
    }
}
```

- **super** refers to the immediate parent class whereas **this** refers to the current class
- **super()** acts as a parent class constructor and should be the first line in a child class constructor
- **this()** acts as a current class constructor (can be used for method overloading)

- What is the difference between **super** and **this**?
- What about **super(...)** and **this(...)**?

```java
package shapes;

public class Shape {
    public String color;

    public Shape(String color) {
        System.out.println("Inside Shape constructor");
        this.color = color;
    }
}
```

```java
public class Rectangle extends Shape {
    public int height;
    public int width;

    public Rectangle(String color) {
        super(color); // => Calling constructor of parent `Shape(String color)`
        System.out.println("Inside Rectangle constructor with one argument");
    }

    public Rectangle(String name, int width, int height) {
        this(name); // => Calling constructor `Rectangle(String color)`
        this.width = width;
        this.height = height;
        System.out.println("Inside Rectangle constructor with three arguments");
    }

    public static void main(String[] args) {
        // Rectangle(3 arguments) => Rectangle(1 argument) => Shape(1 argument)
        Rectangle r = new Rectangle("red", 10, 20);
        System.out.println(r.color);
        System.out.println(r.width);
        System.out.println(r.height);
    }
}
```

- **super** refers to the immediate parent class whereas **this** refers to the current class
- **super()** acts as a parent class constructor and should be the first line in a child class constructor
- **this()** acts as a current class constructor (can be used for method overloading)

What are **static fields** and **methods**?

Static fields are variables that are common and available to all instances of a Class

Methods are a block of code that perform a task. You can think of them as functions of a class.

```java
package circle;

public class Circle extends Object {
    // Every class extends Object, it is not needed though
    private static final double pi = 3.14159;
    private int x, y;
    private int r;

    // Only 1 variable for all Circle objects
    static int no_circles = 0;

    public Circle() {
        super(); // not needed
        no_circles++;
    }

     public double circumference() {
        return 2 * pi * r;
    }
}
```

Documentation

JavaDoc

Documentation

- Why is documentation important? When should you use it
- What does the term "self-documenting" code mean?
  - **Code that documents itself**. It is readable inherently. Usually accomplished through variable name and function names
- When can comments be bad (code smell)?
  - Comments become **stale** & does not get updated with new changes
  - Possibly hinting that your design/code is **too complex**

Comments in Java

Single Line

```
// Single line comment
```

Multi-line comment

```
/**
 * This is multi-line
 * documentation
 */
```

JavaDoc

JavaDoc is one way of documenting in Java.

It mainly targets class definitions and method/function definitions.

in COMP2511, you will not have to use JavaDoc documentation unless asked.

JavaDoc

```java
/**
 * File class that stores content under a file name
 */
public class File {
    /**
     * Constructor used to create a file
     * @param fileName the name of the file
     * @param content contents of the file
     */
    public File(String fileName, String content) {}

    /**
     * Constructor used to make a partial file when receiving a new file
     * I.e., content.length() != fileSize with no compression
     * @param fileName
     * @param fileSize
     */
    protected File(String fileName, int fileSize) {}

    /**
     * Checks if transfer has been completed
     * @return true if it has been completed
     */
    public boolean hasTransferBeenCompleted() {}
}
```

Inheritance

What is it?

In Java, a class can inherit attributes and methods from another class. The class that inherits the properties is known as the sub-class or the child class. The class from which the properties are inherited is known as the superclass or the parent class.

Known as a "is-a" relationship

Code Demo

Employee.java & Manager.java

Code Demo

1. Create a **Employee** class with a **name** and **salary**
2. Create setters & getters with JavaDoc
3. Create a **Manager** class that inherits **Employee** with a **hireDate**
4. Override **toString()** method

Code Demo

The java extension packs come with some features you can use to generate boilerplate code

Code Demo

How many constructors does a class need?

Technically none. If a class is defined without a constructor, Java adds a default constructor

However, if a class needs attributes to be assigned (e.g., has a salary), then a constructor must be assigned

```java
package employee;

import java.time.LocalDate;

public class Employee {
    private String name;
    private double salary;

    /**
     * Creates an Employee with the given name and salary
     * @param name The full name of the employee
     * @param salary The employee's yearly salary
     */
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    /**
     * Gets the name of the Employee
     * @return full name of employee
     */
    public String getName() {
        return this.name;
    }

    /**
     * Sets the name of the Employee
     * @param name new full name
     */
    public void setName(String name) {
        this.name = name;
    }

    public double getSalary() {
        return this.salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return getClass().getName() + "[name=" + name + ", salary=" + salary + "]";
    }
}
```

```java
package employee;

import java.time.LocalDate;

public class Manager extends Employee {
    private LocalDate hireDate;

    /**
     * Creates a manager with a given name, salary and hiring date
     * @param name
     * @param salary
     * @param hireDate
     */
    public Manager(String name, double salary, LocalDate hireDate) {
        super(name, salary);
        this.hireDate = hireDate;
    }

    public LocalDate getHireDate() {
        return this.hireDate;
    }

    @Override
    public String toString() {
        return super.toString() + "[hireDate=" + hireDate + "]";
    }

}
```

Access Modifiers

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Default

The default access modifier is also called **package-private**, which means that all members are visible within the same package but aren't accessible from other packages

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Public

It is accessible to everything. The least restrictive modifier.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Private

It is accessible only to the same class (not including main). The most restrictive modifier.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Protected

Can be accessed in the same package and in inheritance.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Streams

Cool Java things

UML Diagram

Controller
LEGEND
Generalisation
Implementation
Composition
Aggreation
Dependency
0..*
GenericSatellite
-linearSpeed: double
-name: String
+GenericSatellite(double linearSpeed, String name)
+getName(): String
SlowSatellite
-speed: double
+SlowSatellite(String name)
FastSatellite
-speed: double
+FastSatellite(String name)

GenericSatellite

```java
package stream;

public abstract class GenericSatellite {
    private double linearSpeed; // in KM/minute
    private String name; // name of satellite

    public GenericSatellite(double linearSpeed, String name) {
        this.linearSpeed = linearSpeed;
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public double getLinearSpeed() {
        return this.linearSpeed;
    }

    @Override
    public String toString() {
        return "{" +
                "name='" + getName() + "'" +
                "}";
    }

}
```

FastSatellite

```java
package stream;

public class FastSatellite extends GenericSatellite {
    private static final double speed = 100.0;

    public FastSatellite(String name) {
        super(speed, name);
    }
}
```

SlowSatellite

```java
package stream;

public class SlowSatellite extends GenericSatellite {
    private static final double speed = 50.0;

    public SlowSatellite(String name) {
        super(SlowSatellite.speed, name);
    }
}
```

Controller

```java
package stream;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Controller {
    private List<GenericSatellite> satellites = new ArrayList<>();

    public void addSatellite(GenericSatellite satelliteToAdd) {
        this.satellites.add(satelliteToAdd);
    }

    public List<GenericSatellite> getSatelliteList() {
        return this.satellites;
    }

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

}
```

Main Function - Streams

```java
public static void main(String[] args) {
    Controller c = new Controller();
    c.addSatellite(new FastSatellite("Fast Satellite 1"));
    c.addSatellite(new FastSatellite("Fast Satellite 2"));
    c.addSatellite(new SlowSatellite("Slow Satellite 1"));

    List<GenericSatellite> allSatellites = c.getSatelliteList();
    System.out.println("All: " + allSatellites);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}, {name='Slow Satellite 1'}]

    // What if I just want FastSatellites only?
    // Method 1, normal for-in/for-each
    List<GenericSatellite> fast1 = new ArrayList<>();
    for (GenericSatellite x : allSatellites) {
        if (x instanceof FastSatellite) {
            fast1.add(x);
        }
    }
    System.out.println("Just fast: " + fast1);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]

    // Method 2, streams
    List<GenericSatellite> fast2 = allSatellites.stream().filter(x -> x instanceof FastSatellite).collect(Collectors.toList());
    System.out.println("Just fast: " + fast2);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]

    // Same, but I typecast at the same time
    List<FastSatellite> fast3 = allSatellites.stream().filter(x -> x instanceof FastSatellite).map(x -> (FastSatellite) x)
            .collect(Collectors.toList());
    System.out.println("Just fast: " + fast3);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]
}
```

```java
public static void main(String[] args) {
    Controller c = new Controller();
    c.addSatellite(new FastSatellite("Fast Satellite 1"));
    c.addSatellite(new FastSatellite("Fast Satellite 2"));
    c.addSatellite(new SlowSatellite("Slow Satellite 1"));

    List<GenericSatellite> allSatellites = c.getSatelliteList();
    System.out.println("All: " + allSatellites);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}, {name='Slow Satellite 1'}]

    // What if im trying to search for something?
    // Look for satellite with name == "Fast Satellite 2"
    // Method 1, normal for-in/for-each
    GenericSatellite g1 = null;

    for (GenericSatellite x : allSatellites) {
        if (x.getName().equals("Fast Satellite 2")) {
            g1 = x;
            break;
        }
    }
    System.out.println(g1);
    // {name='Fast Satellite 2'}

    // Method 2, streams
    GenericSatellite g2 = allSatellites.stream().filter(x -> x.getName().equals("Fast Satellite 2")).findFirst().orElse(null);
    System.out.println(g2);
    // {name='Fast Satellite 2'}

    // Now I search for "Fast Satellite 3", which doesn't exist
    GenericSatellite g3 = allSatellites.stream().filter(x -> x.getName().equals("Fast Satellite 3")).findFirst().orElse(null);
    System.out.println(g3);
    // null
}
```

Feedback

https://forms.gle/fZDe2zhbo52UNnwh7