Assess Yourself

A world class chef is working to develop a robotic assistant, and they've hired you to build a simulated kitchen to test it out.

The robot needs to be able to:

- Use normal human tools and utensils (oven, fork)
- Open things (cupboards and containers)
- Prepare ingredients (cutting, dicing, boiling)
- Be operated by a human (for safety reasons)

How would you develop this system? What classes would you use? What methods and attributes would they have? What interface does your program have between the user and the robot?

Assess Yourself

Class: `Ingredient`

- Attributes
  - position
  - orientation
  - size
  - type
- Methods

Assess Yourself

Class: `Tool`

- Attributes
  - position
  - orientation
  - size
  - type
- Methods
  - getGraspPosition
  - operate

Assess Yourself

Class: `Container`

- Attributes
  - position
  - orientation
  - size
  - contents
- Methods
  - fill
  - empty

Assess Yourself

Class: `Robot`

- Attributes
  - tools
  - ingredients
  - containers
  - recipe
- Methods
  - open
  - close
  - use
  - prepare
  - findPath

SWEN20003
Object Oriented Software Development

Privacy and Mutability

Semester 1, 2019

Gentle Reminder

Grok worksheets are **not** assessed.

But...

You are expected to complete **all** of them; everything in the worksheets is examinable (except Grok specifics like `grok.Input`).

If you ask questions that could be answered by completing the worksheets, expect the tutors to ask "have you completed the worksheets?" and walk away if the answer is "No".

The Road So Far

- OOP and Java Foundations
    - Classes and Objects
    - Strings and Wrappers
    - Formatting
    - Methods and Abstraction
    - Input and Output
    - Arrays

Lecture Objectives

After this lecture you will be able to:

- Correctly use privacy modifiers
- Explain the concepts of **information hiding**, **access control** and **encapsulation**
- Describe **mutability**
- Describe a **privacy leak**
- Identify and avoid privacy leaks in your programs

In-class code found here

Privacy?

```java
public class Movie {

    public String title;

    public Movie(String title) {
        this.title = title;
    }
}
```

```java
Movie movie = new Movie("The Avengers: Endgame");
```

```java
movie.title = "Apple: Technology Gone Wrong";
```

Mutability

Keyword

Mutable: An object is **mutable** if **any** of its instance variables can be **changed** after being initialised.

Keyword

Immutable: An object is **immutable** if **none** of its instance variables can be changed after being initialised.

Privacy

Keyword

Information Hiding: Using privacy to “hide” the details of a class from the outside world.

Keyword

Access Control: Preventing an outside class from **manipulating** the properties of another class in **undesired** ways.

Keyword

Encapsulation: Using **privacy** to **restrict unwanted access** to the **values** of an object.

Privacy?

How can we use **information hiding** and **access control** to prevent this from happening?

```java
Movie movie = new Movie("The Avengers: Endgame");
```

```java
movie.title = "Apple: Technology Gone Wrong";
```

```java
public class Movie {

    public String title;

    public Movie(String title) {
        this.title = title;
    }
}
```

Privacy Modifiers

Keyword

Private: Only available to methods defined in the same class; should be applied to all (mutable) instance variables, and some methods.

Keyword

Package: Default, applied if nothing else specified; available to all classes in the same *package*.

Privacy Modifiers

Keyword

Protected: Available to all classes in the same package *and also* to any subclasses that inherit from the class.

Keyword

Public: Available at all times, everywhere.

Privacy Modifiers

Modifier	**Class**	**Package**	**Subclass**	**World**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`default`	Y	Y	N	N
`private`	Y	N	N	N

Privacy?

```java
public class Movie {

    private String title;

    public Movie(String title) {
        this.title = title;
    }
}
```

```java
movie.title = "Apple: Technology Gone Wrong"; // No longer valid
```

- Fixed... sort of
- Can no longer access/use/retrieve `title`
- What now?

Getters and Setters

Keyword

Getter/Accessor: A method that returns an instance variable from a class.

```
public <type> get<VarName>() {
    return var;
}
```

Keyword

Setter/Mutator: A method that modifies the value of an instance variable.

```
public void set<VarName>(<type> var) {
    this.var = var;
}
```

Mutability II

Keyword

Immutable: A class is **immutable** if all of its attributes are private, and it contains no setters.

Getters and Setters

```java
public class Actor {
    public int age;
}
```

Why is this implementation **dangerous**?

```java
Actor actor = new Actor();
```

```java
actor.age = -42;
```

Getters and Setters

Solution #1:

```
public class Actor {

    private int age;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```
actor.setAge(-42); // Still possible to set bad values
```

Getters and Setters

Solution #2:

```java
public class Actor {

    private int age;

    public int getAge() {
        return age;
    }

    // Implementing access control and validation
    public void setAge(int age) {
        if (age <= 0) {
            return;
        }
        this.age = age;
    }
}
```

Getters and Setters

Solution #3:

```java
public class Actor {

    private int age;

    public int getAge() {
        return age;
    }

    // Implementing access control and validation
    public void setAge(int age) {
        if (age <= 0) {
            throw new InvalidAgeException(age);
        }
        this.age = age;
    }
}
```

Getters and Setters

Invalid:

```java
Actor actor = new Actor();
actor.age = 10;
```

Valid:

```java
Actor actor = new Actor();
actor.setAge(-10); // No effect
actor.setAge(10);
```

Assess Yourself

```java
public class Movie {

    private String title;

    public Movie(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Assess Yourself

```java
public class Member {

    private Movie favouriteMovie;

    public Member(Movie favouriteMovie) {
        this.favouriteMovie = favouriteMovie;
    }

    public Movie getFavouriteMovie() {
        return favouriteMovie;
    }

    public void setFavouriteMovie(Movie favouriteMovie) {
        this.favouriteMovie = favouriteMovie;
    }
}
```

Assess Yourself

```java
Movie movie = new Movie("Iron Man");
Member matt = new Member(movie);
System.out.println(matt.getFavouriteMovie().getTitle());

Member ellie = matt;
ellie.setFavouriteMovie(new Movie("Copper Man"));
System.out.println(matt.getFavouriteMovie().getTitle());
```

```
Iron Man
Copper Man
```

Objects are references!

Both `ellie` and `matt` refer to the same object.

Standard Methods - *Copy Constructor*

```
public <ClassName>(<ClassName> var) {
    <block of code to execute>
}
```

- Creates a **separate copy** of the object sent as input

```
public Actor(Actor actor) {
    this.firstName = actor.firstName;
    this.lastName = actor.lastName;
    ...
}
```

Assess Yourself

```java
public class Member {

    private Movie favouriteMovie;

    public Member(Movie favouriteMovie) {
        this.favouriteMovie = favouriteMovie;
    }

    public Member(Member member) {
        this.favouriteMovie = member.favouriteMovie;
    }

    public Movie getFavouriteMovie() {
        return favouriteMovie;
    }

    public void setFavouriteMovie(Movie favouriteMovie) {
        this.favouriteMovie = favouriteMovie;
    }
}
```

Assess Yourself

```
Movie movie = new Movie("Iron Man");
Member matt = new Member(movie);
System.out.println(matt.getFavouriteMovie().getTitle());

Member ellie = new Member(matt);
ellie.setFavouriteMovie(new Movie("Copper Man"));
System.out.println(matt.getFavouriteMovie().getTitle());
```

```
Iron Man
Iron Man
```

Problem solved. When assigning `matt` to `ellie`, we create a **copy** so we have two distinct/separate objects.

Assess Yourself

```
Movie movie = new Movie("Iron Man");
Member matt = new Member(movie);
System.out.println(matt.getFavouriteMovie().getTitle());

Member ellie = new Member(movie);
Movie newMovie = ellie.getFavouriteMovie();
newMovie.setTitle("Copper Man");
System.out.println(matt.getFavouriteMovie().getTitle());
```

```
Iron Man
Copper Man
```

Objects are still references!

We created `ellie` and `matt` using the **same** `Movie` object!

Pitfall: Privacy Leaks

Keyword

Privacy Leak: When a reference to a private instance variable is made available to an **external** object, and unintended/unknown changes can be made.

Pitfall: Copying

- When **assigning** an object, make sure to assign a **copy**!
- When **returning** an object, make sure to return a **copy**!
- When copying an object, make sure to create copies of the objects **inside it** too! This is called a **deep copy**.

Mutability III

Keyword

Immutable: A class is **immutable** if all of its attributes are private, it contains no setters, and only returns **copies** of its (mutable) instance variables.

Final Solution

```java
public class Member {

    private Movie favouriteMovie;

    public Member(Movie favouriteMovie) {
        this.favouriteMovie = favouriteMovie;
    }

    public Member(Member member) {
        this.favouriteMovie = member.getFavouriteMovie();
    }

    public Movie getFavouriteMovie() {
        return new Movie(favouriteMovie);
    }

    public void setFavouriteMovie(Movie favouriteMovie) {
        this.favouriteMovie = favouriteMovie;
    }
}
```

Metrics

Build on our IMDB clone project by adding the `Member` class, ensuring that you follow proper **information hiding** principles. You may add whichever attributes and methods you feel are appropriate for the problem.

To make your system more “secure”, make the `Member` class **immutable**, so that once it has been created, it cannot be modified.