**Week 10 Lab – Polymorphism**

# Lab Intro

**Key Term – Polymorphism**Polymorphism means *many forms*. When combined with inheritance, it can lead to less duplicated code in different situations (e.g. where the behaviour (methods) of different related classes have the same name, but execute different logic).



In the first webinar and lecture this week, we saw another very powerful object-oriented concept: **polymorphism**. In this lab, we will implement programs that make use of this concept in order to make our programs more reusable and extensible. We will also get some more experience with using access modifiers.

Launch IntelliJ and create a new **Project** named **Week 10**.

## Learning Outcomes

* **Implement** polymorphic behaviour to reduce code duplication
* **Overload** constructors and methods to provide different ways of achieving object creation and behaviour
* **Enforce** data hiding through the use of access modifiers

## Resources

* Week 10 lecture slides

**Key Term – Method Overloading**A method is referred to as an overloaded method if there are multiple methods in the same class with an identical name but a different **signature**. A method’s **signature** is the combination of the method name and the order and types of the parameters in the parameter list.



**Exercise 1 – Method Overloading**

We know that we can define multiple methods in the class with the same name, provided that they have different **signatures** – such methods are referred to as **overloaded methods**.

1. Create a projectMethod named **MethodOverloadDemo** with a main method (you’ll come back to this class shortly)
2. Create a second class named **PatternGenerator** (no main method)
   1. This class should have one member variable named **patternChar** which should be of the **char** data type, and has a default value of ‘@’ (**Note**: characters must be enclosed in single quotes ‘ ‘, double quotes are used for Strings).
   2. Provide the **four overloaded methods below – they should be void (since they won’t return anything to the calling code)**
      1. **printPattern(int rows)** – this method simply prints the pattern (where **rows** is the number of times it should print – you will want a println here)
      2. **printPattern(int rows, char patternChar)** – this method allows the user to pass in a character which is then used to print the rows
      3. **printPattern(int rows, int cols)** – this method prints both rows and columns of the default character
      4. **printPattern(int rows, int cols, char patternChar)** – this method prints both rows and columns of the character supplied to it
3. Go back to your class with the main method, and create an object of the **PatternGenerator** class and call the different **printPattern** methods.
   1. Figure 1 shows an example of calling the **printPattern** method four times
      1. passing in **4** for **rows**
      2. passing in **5** for **rows** and **5** for **cols**
      3. passing in **2** for **rows** and **15** for **cols**, and a **patternChar** of **‘!’**, and
      4. passing in **3** for **rows** and **‘#’** for **patternChar**

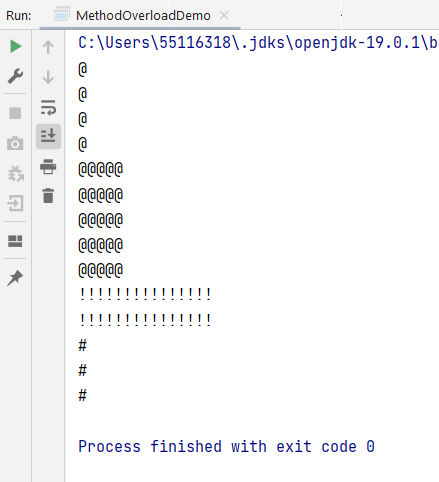


Figure Exercise 1 Example Output

# Exercise 2 – Constructor Overloading

This exercise will involve creating multiple constructors in a class, and reducing the duplication across those constructors by getting them to call each other.

**Key Term – Constructor Overloading**Like methods, constructors can also be **overloaded**. An overloaded constructor is distinguished by just the order and types of its parameters – the name is not part of the signature (since all constructors in a class have the same name – the name of the class).



1. Create a class named **ConstructorDemo** with a **main** method, which you shall revisit shortly
2. Create a class called **Movie** (no main method), with the following in it:
   1. **Variables**:
      1. String name (the name of the movie)
      2. double imdbRating (rating of a movie according to the Internet Movie Database)
      3. int lengthMinutes (the length of the movie in minutes)
      4. int yearReleased (the year the movie was/is going to be released)
   2. **Constructors** – provide four overloaded constructors in this class, each of which accepts different order/types of parameters, the order and types being depicted in Figure 2.
   3. Constructors can delegate work to each other to reduce duplication via the **this** keyword. e.g. the fourth constructor in Figure 2 can call the third constructor (which accepts a String name, double imdbRating, and an int lengthMinutes), and then initialises the fourth parameter (yearReleased) itself. Implement this logic now – get the constructors to delegate work to each other using the ***this*** keyword (see lab slides for an example of this).
   4. Finally, in this class, provide a method named **void** **printMovieDetails()** that simply prints out the values of the four instance variables
3. In your **ConstructorDemo** class, create an ArrayList of Movies and add to it **four movie objects** – each of which should call one of the four constructors defined in the Movie class.
4. Loop through the ArrayList and call each of the object’s **printMovieDetails** method
5. You will notice that some of the objects created (e.g. when passing in just a String for the name), have default values for the other attributes (e.g. default for imdbRating will be 0.0, as it’s a float). We will resolve this in the next exercise when we implement getters and setters.

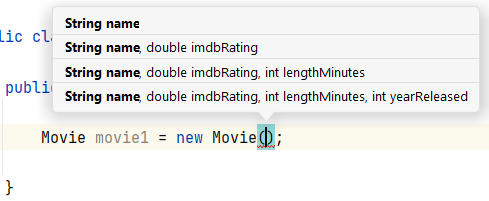
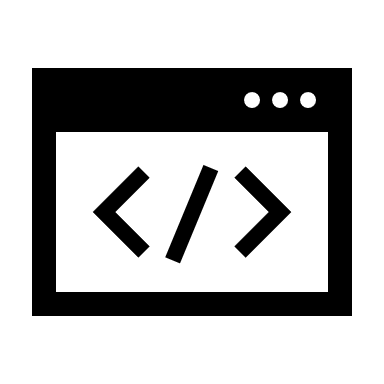


Figure IntelliJ Parameter Reminder

Save for later – export to Zip and save on Onedrive



# Exercise 3 – Polymorphic ArrayList

In Webinar 10A, we saw an example of an array that exhibited polymorphism. We refactored existing code that previously had three different arrays to store **Squares**, **Circles**, and **Stars**. We swapped these out for a single array to store references to **Shape** objects – the superclass of Square, Circle, and Star. This type of Polymorphism is known by several names (e.g. **Inclusion Polymorphism**, **Subclass Polymorphism**, or **Subtype** **Polymorphism**) and is achieved when we have an inheritance hierarchy.

For this exercise, it will be less descriptive, you will be given a UML diagram of some classes, and you should create an ArrayList that exhibits Polymorphism (as we did in Webinar 10A).

1. Implement the three classes in the UML diagram in Figure 5. You should implement the variables and the methods (including the logic you would expect to find in those methods). Feel free to add your own variables and methods as appropriate.
2. After implementing the three classes in the UML diagram, create a class with a main method that creates some objects of both the **Word** and **PowerPoint** classes, and test the functionality accordingly.
3. Create a class called **Directory**, which maintains an ArrayList of **File** objects (ArrayList<File>). Add methods to the Directory class to allow someone to add/remove a file to the directory (ArrayList), and a method that loops through all of the **File** objects and calls each of their **printFileInfo** method
4. Create a Directory object and add the previously created Files (Words and PowerPoint files) to the ArrayList of the directory. Loop through the directory’s file ArrayList and call each file object’s **printFileInfo** method

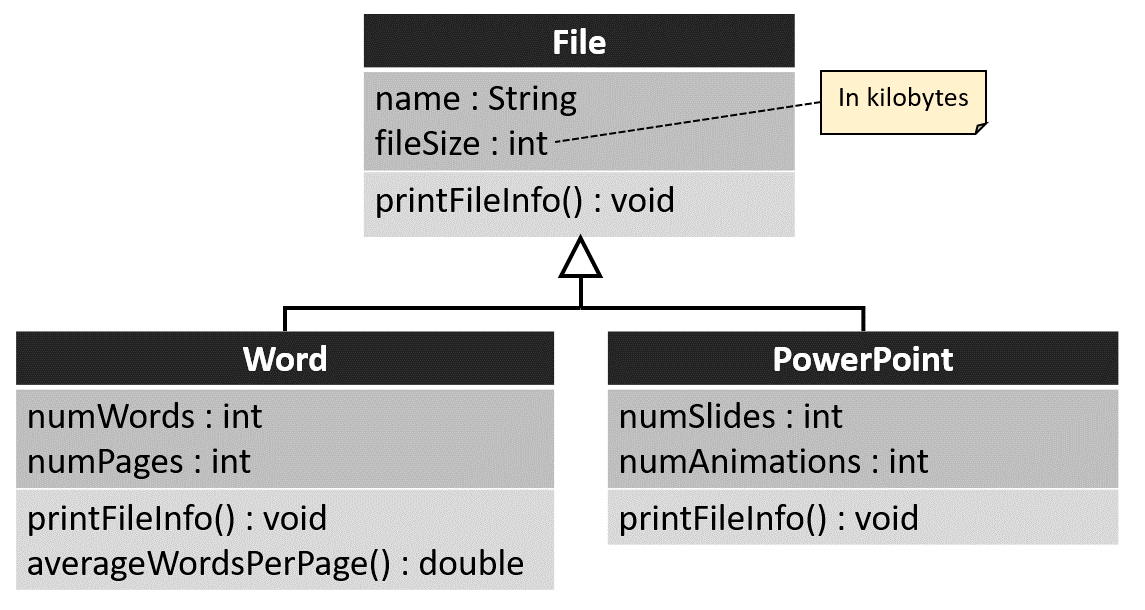
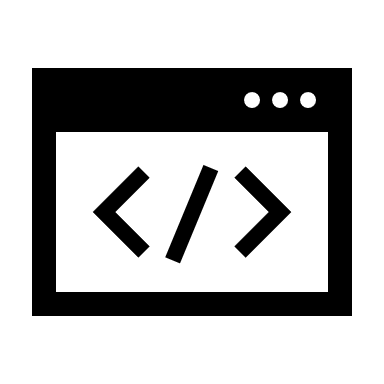


Figure Exercise 4 UML Diagram

# Exercise 4 – Data Hiding



This exercise does not really relate to Polymorphism, but it does relate to the concept of **data hiding** we saw in **Webinar** **9B**, where we should make instance variables private, and only provide public methods to access those variables if the user needs access to them.

**Key Term – Getters and Setters**

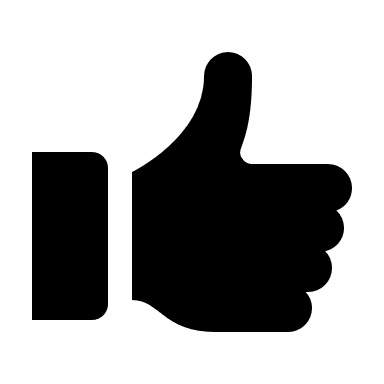
Getters and setters refer to public methods that allow access to private variables. Most IDEs are capable of generating them using a built-in shortcut. Getters and Setters help enforce data hiding and ensure our classes are **encapsulated**.



For example, if we want someone to be able to retrieve a value for one of the variables in the Movie class, we can provide a **public** method that simply returns that variable. A method that gets a variable in this way is typically called a **getter** (e.g. if we want someone to be able to get the name of a movie, we would provide a method that **returns** a String object (the name variable).

**Good Programming Practice – Getters and Setters**

It is considered good practice to always make instance variables private, and if they need to be accessed from outside of the class, provide public methods to get/set them. If a variable needs to be read-only, you can just provide a **get** method to retrieve it, but not a **set** method.



If we want someone to be able to set one of the variables in the Movie class, we can provide a **public** method that **sets** the variable to a new value. This type of method is typically called a **setter** (e.g. if we want to provide a way for someone to change the name of the movie, we can define a **void** method called **setMovieName,** which takes in one parameter – the new name of the movie.

Because making instance variables private and providing public methods to access them is so common, most IDEs like IntelliJ have shortcuts to make this process less tedious – we’ll see this in just a second.

1. In your **Movie** class from Exercise 2, make all of the instance variables **private.**
2. While in your Movie class, hold down **ALT** and press the **INSERT** key to generate the menu items which we saw last week, and select **Getter and Setter** (Figure 3).
3. Generate **Getters and Setters** for all four fields (Figure 4)
4. Incorporate validation checking in the setter methods, as defined in Table 1.
5. Test that the validation works by calling the methods with invalid input (e.g. calling setImdbRating(12.3) should not result in the imdbRating changing)
6. Ensure the objects have valid values for each of their fields

|  |  |
| --- | --- |
| Figure Generating Getters and Setters in IntelliJ (ALT+INSERT) | Figure Selecting Field Getters and Setters |

Table Validation to be Performed by Movie Setter Methods

|  |  |
| --- | --- |
| **Method** | **Validation Required** |
| setName | The name passed in must be between 1 and 100 characters (**Hint:** all strings have a .length() method that returns the number of characters in the String.) For each of these methods, you should print out if the validation fails. |
| setImdbRating | The rating supplied must be between 0 and 10.0 inclusive. |
| setLengthMinutes | The length of the movie must be between 0 and 240 inclusive. |
| setYearReleased | The year must be between 1900 and 2024. |

# Next Week’s Lab Session

In next week’s lab session, we will have some time set aside to provide formative feedback on the progress you’ve made on your game assessment so far. If you have not started your assessment, now is the time to do so. There are many places you could start, but you’ll likely begin with defining a set of classes to represent the different *things* inside your game. The lecture from week 9 (Object-Oriented Analysis & Design) will be helpful if you’re struggling to identify classes to implement in your game.