Lab 1: Basic Object-Oriented Programming

Aims

1. Get used to the environment and compiling Java using IntelliJ.
2. Implement a simple example of object-oriented programming.
3. Working with existing code (some IntelliJ tips).

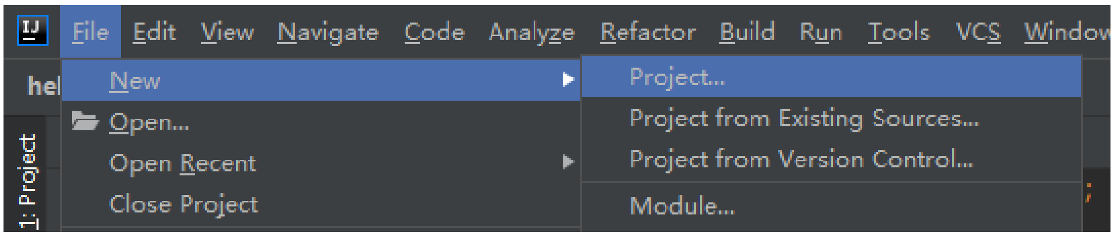
Students may have different levels of experiences with the IDE and OOP. Feel free to jump in the section which suits your abilities.

IntelliJ Basics

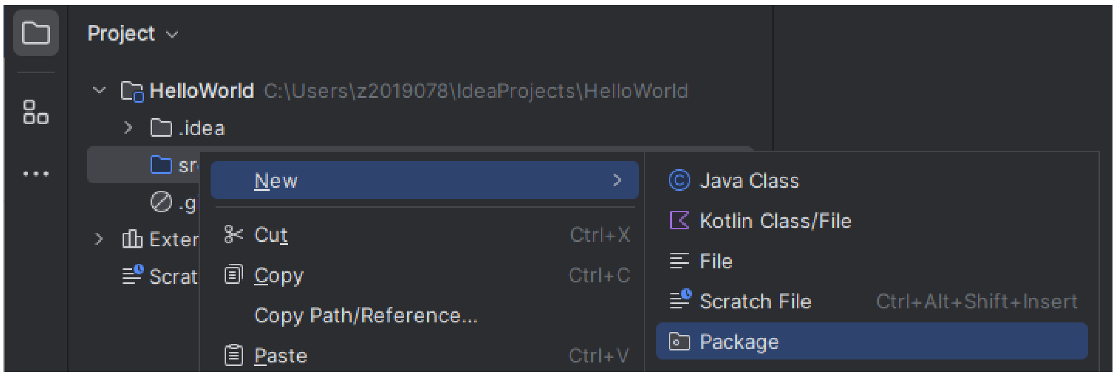
To develop maintainable software, one needs to be familiar with modern integrated development environment (IDEs). This module will use Java and IntelliJ (v2024.1.4). The first part of this lab session is a guide through a simple "Hello World" program for refreshment.

To start, open the IntelliJ, and create a new Java project:

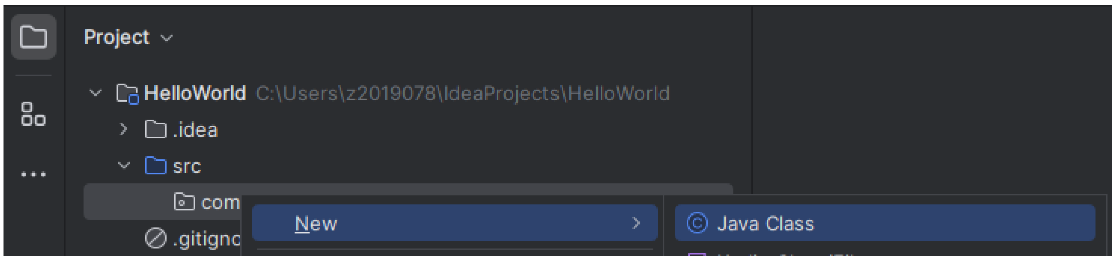
- File – New – Project, then Select "Java" in the left column.
- Do not "add sample code".
- Name the project as "HelloWorld", make sure no space within words.
- Verify the default workspace folder location, change the location if needed.



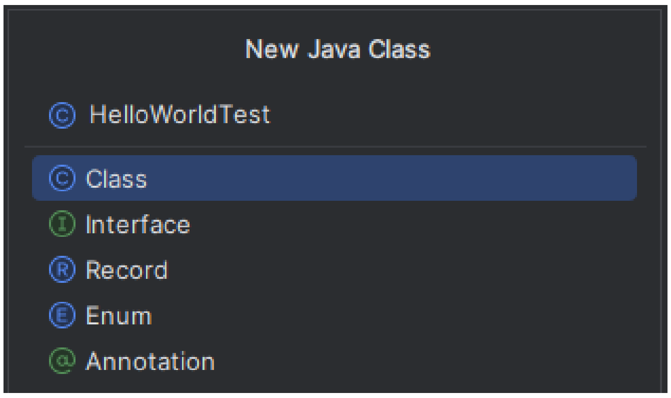
The created project should appear in the "Project" panel on the left hand side. Note that no file has been yet created. Add a source file by right click on the "src" folder and select "New – Java Class". But this would create a class wildly existing. For now, let's first create a package to put the new class in. Then create the class. Do right click on the "src" folder, and select "New – Package". Give the package a name, such as "com.lab1".



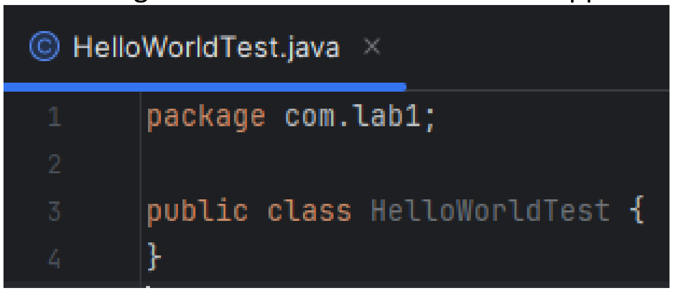
Right click the “com.lab1” package, select New – Java Class.



In the dialog box, add a name (e.g., “HelloWorldTest”), and double click the blue “Class” item.

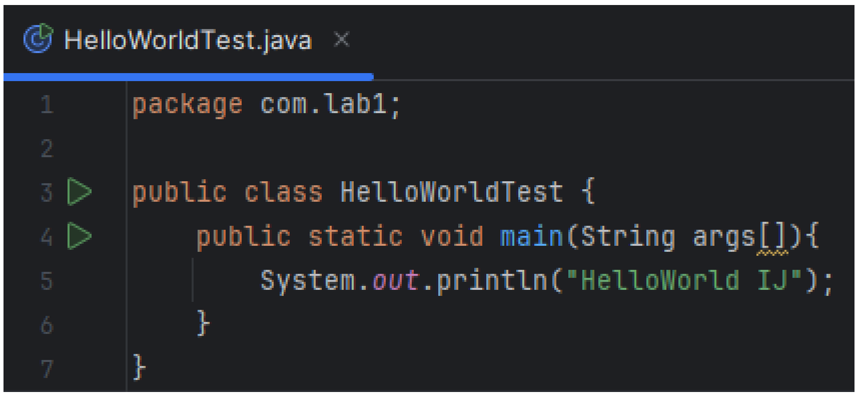


The editing window for this class will then appear.



```
package com.lab1;

public class HelloWorldTest {
}
```

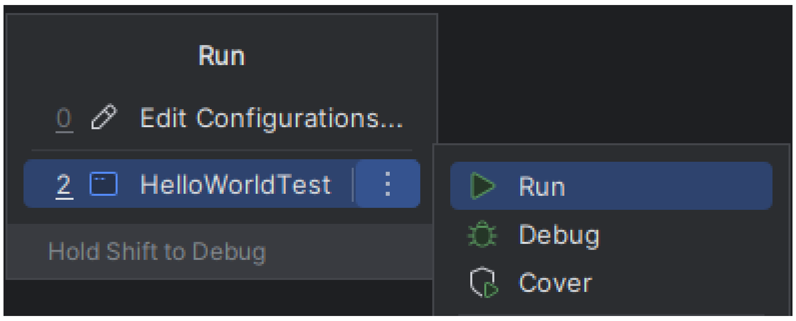
Insert some lines of code as below. In the Menu bar, click “Build – Build Project”. Then “Run – Run ‘HelloWorldTest.java’”.



HelloWorldTest.java

```java
package com.lab1;

public class HelloWorldTest {
    public static void main(String args[]){
        System.out.println("HelloWorld IJ");
    }
}
```

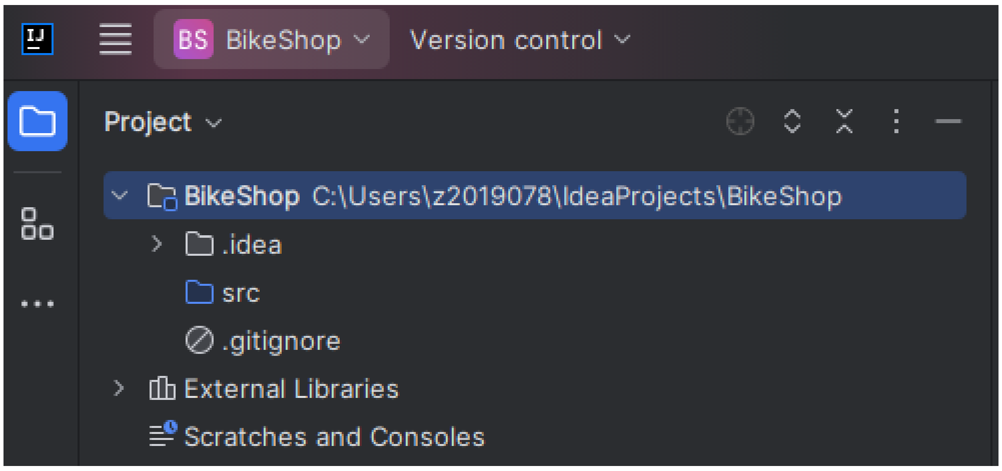
Note that for the first run, you may need to instead use “Run – Run…” option. Then choose the ‘HelloWorldTest’ configuration to run.



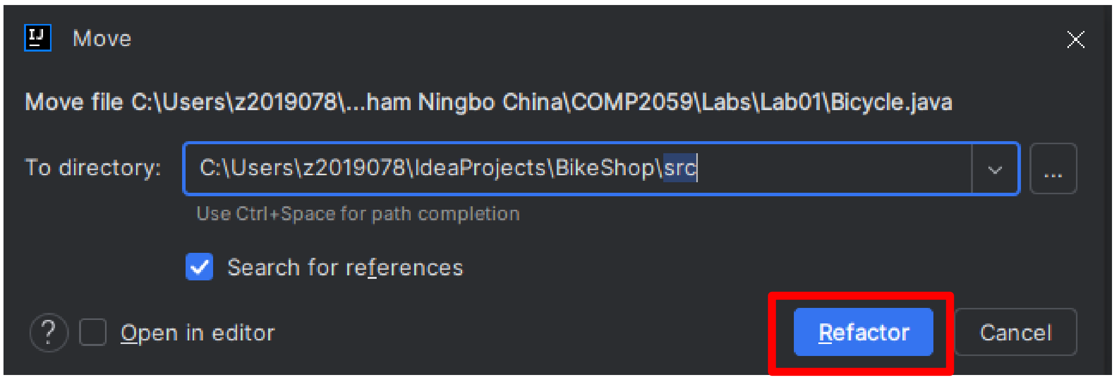
When you see the “Hello World IJ” appear in the console window, you are ready for the next part of the lab.

OOP Example

Create a new Java project, name it as “BikeShop”. Note: do not tick the “Add sample code”, so that the “src” folder appear as empty.



Download the code “Bicycle.java” from Moodle page and import the source code into your project. To do this, drag and drop the “Bicycle.java” file into the src folder. A dialog will show as below. Click “Refractor”.



Move

Move file C:\Users\z2019078\...ham Ningbo China\COMP2059\Labs\Lab01\Bicycle.java

To directory: C:\Users\z2019078\IdeaProjects\BikeShop\src

Use Ctrl+Space for path completion

☑ Search for references

☐ Open in editor

As a better practice, you might need to manually create the “bike” package in “src” folder, and drag the “Bicycle.java” into this new package.

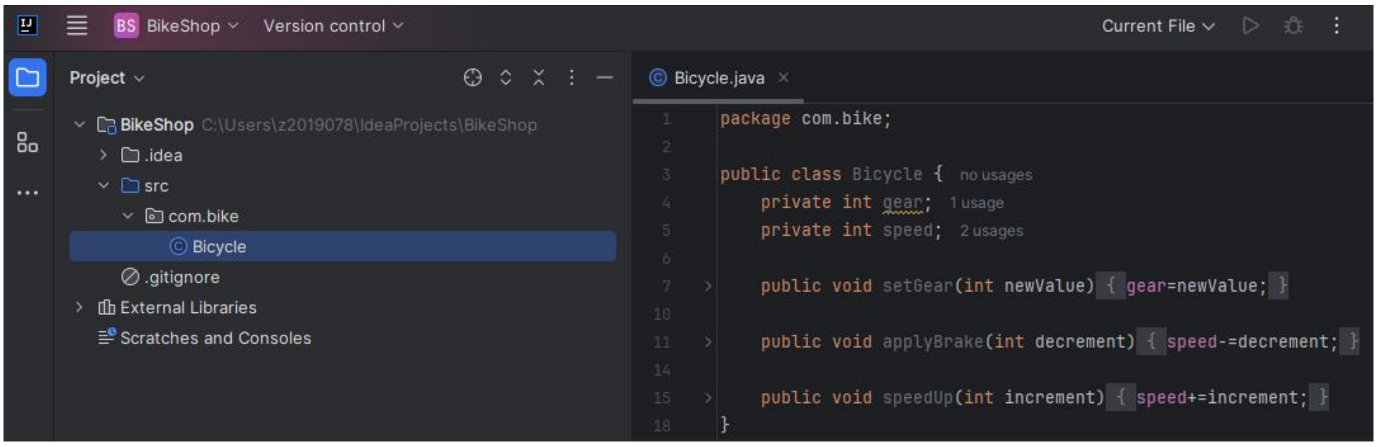
===

Questions:

- Compare those two approaches (with package and without package), do you find any difference in the “Bicycle.java” code after pressing the “Refactor” button? Do both and check the code of “Bicycle.java”.
- What does “Refactor” mean? After the comparison above, can you make a guess about its purpose? Why isn’t the button (enclosed in red) named otherwise, like “Move”, or “Import”?

===

To continue, let’s use the code version with package, shown as below.



```
package com.bike;

public class Bicycle {   no usages
    private int gear;   1 usage
    private int speed;   2 usages

    public void setGear(int newValue) { gear=newValue; }

    public void applyBrake(int decrement) { speed-=decrement; }

    public void speedUp(int increment) { speed+=increment; }
}
```

The project should compile, but there is no main function so nothing will appear (or ClassNotFoundException appears) if you try to run it.

===
Task: Extend the BikeShop project.

a. Write a main class (“Main.java”) and main function which instantiates a Bicycle. Try changing the speed.
b. Write a suitable constructor for the “Bicycle” class so you can set the initial gear and speed values.
c. Add a method called “switchLight” which turns the light on if it is off, and vice versa.
d. Add a method to print out the current speed, gear, and light state.

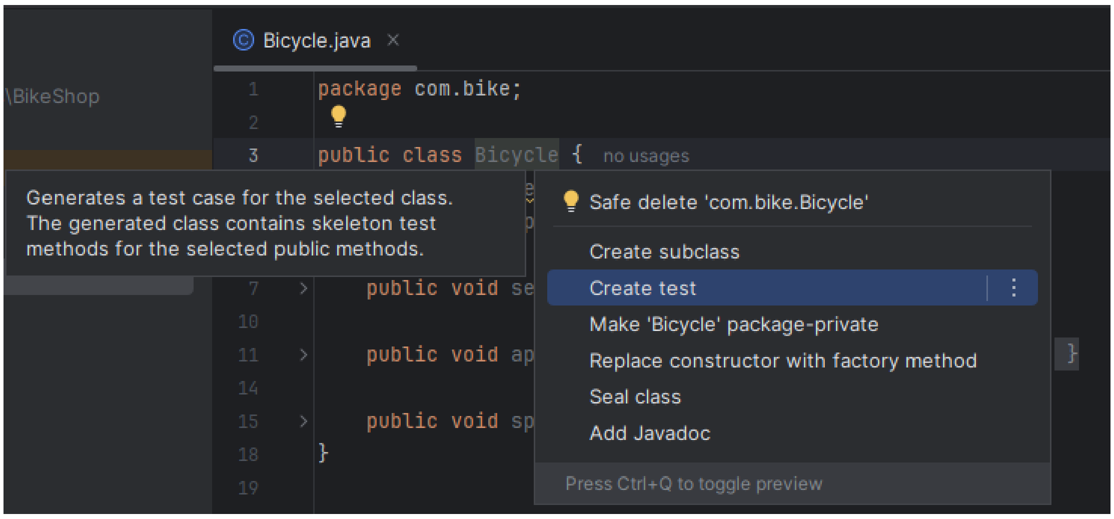
===

Tip: In IntelliJ, automating the generation of the constructor, getter, setter, toString, etc., can be done from “Code – Generate…” (with cursor selecting the class name), then in the Generate popup, choose the one you need.

Now, let’s suppose that a class for mountain bike is needed which has specific features associated with it. So, it makes sense to create a subclass of a bicycle, by inheriting from Bicycle.

To do this, you can create a class named “MountainBike”, probably in the same package, then handcode the lines using the keyword “extends” etc.

Alternatively (**Tip alert**), use shortcut provided in IntelliJ. Move the mouse over its super class’s name, “Bicycle”, press “Alt+Enter” (or “Option+Return” for Mac), then the following menu shows:



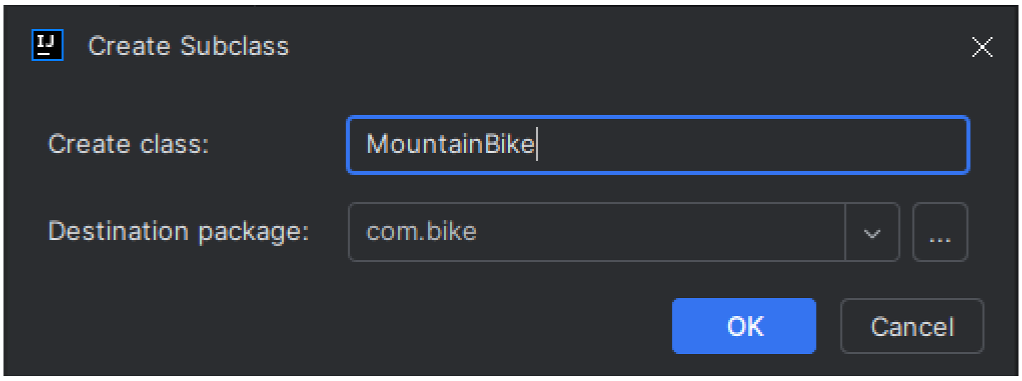
```
package com.bike;

public class Bicycle {  no usages
```

Generates a test case for the selected class. The generated class contains skeleton test methods for the selected public methods.

Bicycle.java
\BikeShop
Safe delete 'com.bike.Bicycle'
Create subclass
Create test
Make 'Bicycle' package-private
Replace constructor with factory method
Seal class
Add Javadoc
Press Ctrl+Q to toggle preview

Tip: This menu can also be found by right click the class name “Bicycle”, then click “Show Context Actions”.

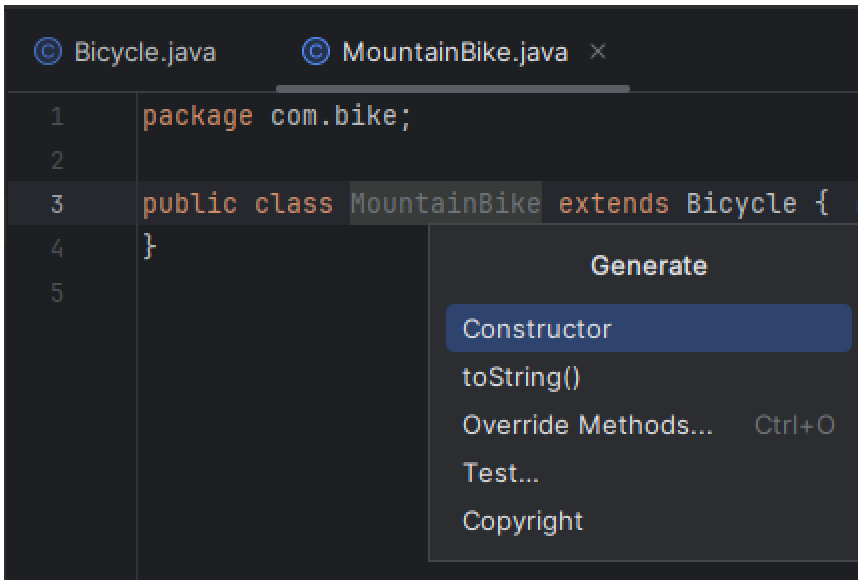
Click on the “Create subclass”, a dialog pops up. Insert the subclass’s name “MountainBike”, then click OK.



Create class: MountainBike

Destination package: com.bike

A subclass named “MountainBike” is now automatically created. You may further choose to Generate the constructor etc.



```java
package com.bike;

public class MountainBike extends Bicycle {
}
```

===

Task:

let’s adds two Boolean variables to store whether a bike has a front suspension and a rear suspension.

a. Modify the constructor of “MountainBike” to take in value for the two suspension variables and set them in the constructor.

b. Add a method called “isFullSuspension” which returns true only if the bike has both front and rear suspension.

c. Create objects for two different mountain bikes, one with full suspension and one without.

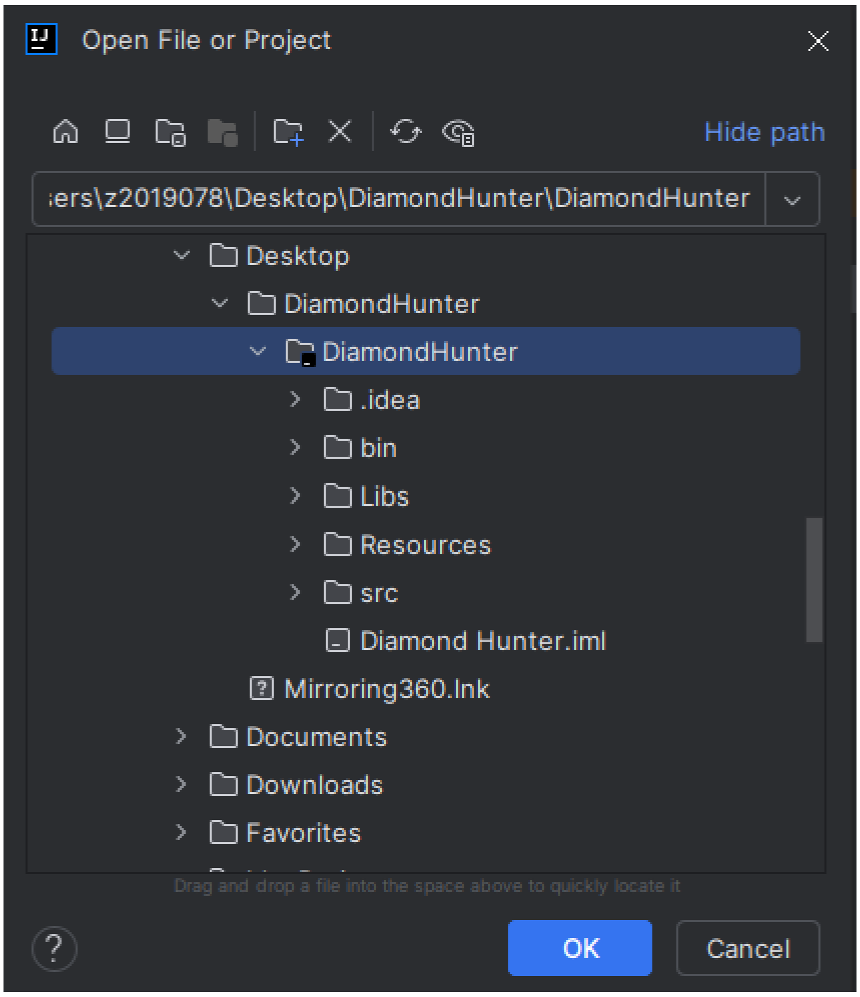
===

Working with existing code

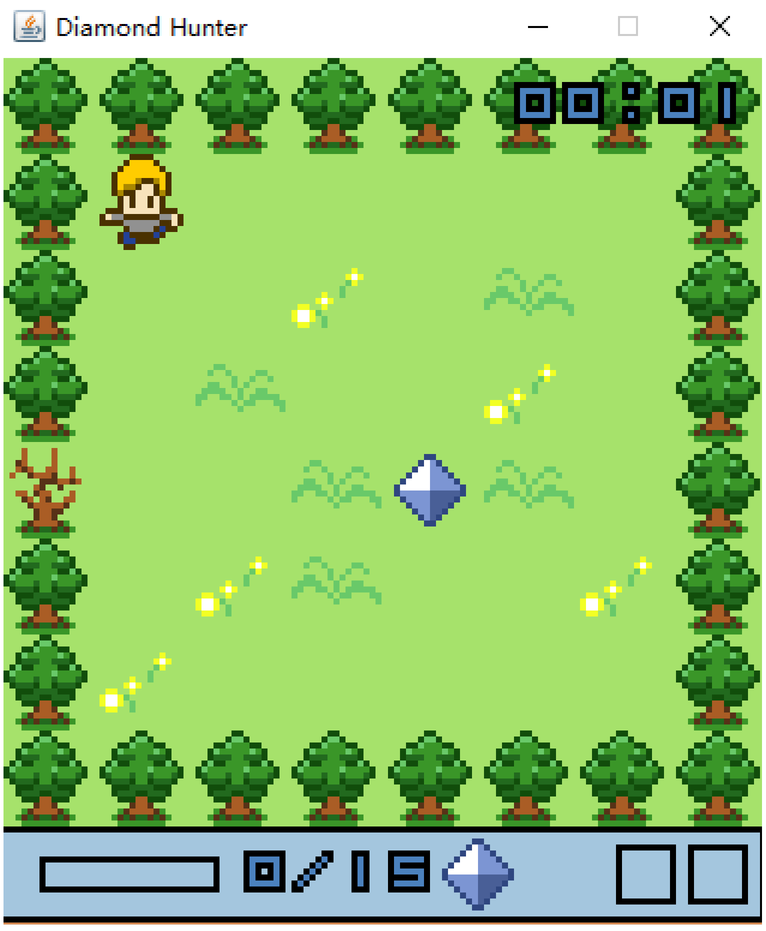
This section will download and explore a larger existing codebase and learn some tips in IntelliJ for navigating project code.

Let's have a look at some real-world source code: a game called "Diamond Hunter". You can find more information about the game here https://www.youtube.com/watch?v=AA1XpWHhxw0

First, download the source code zip file ("DiamondHunter.zip") from Moodle, extract it to the desktop. In IntelliJ, choose "File – Open", and locate the directory as below. Then click OK. Note that in this zip file, the project is already an IntelliJ project, so you just need to open it.



Before building the project, go to "File – Project Structure – Project Settings – Project", and check the Project SDK and Language Level. Choose version 22 by default (**Tip alert**). Then Run the project, and you will see the following game window.



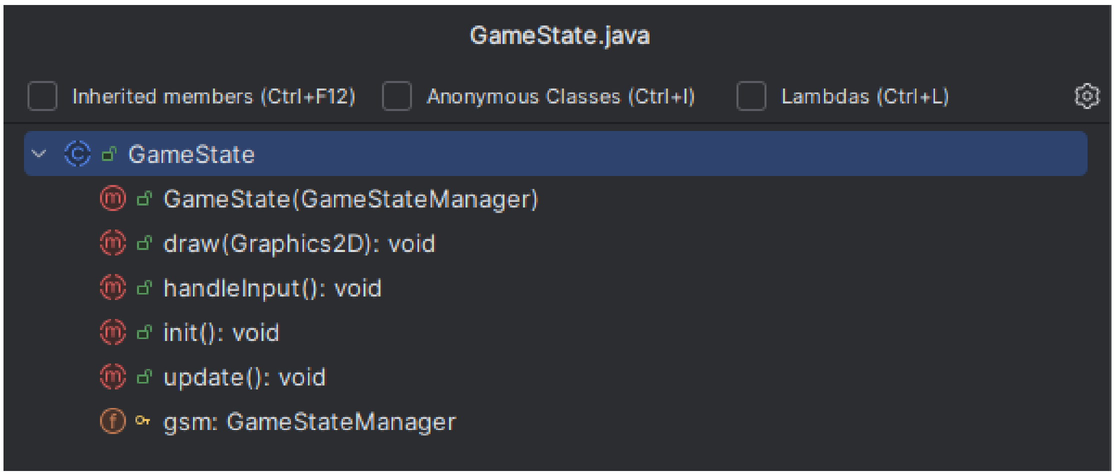
Tip: Check the "File | Project Structure | Project Settings | Project | Project SDK" and "Run | Edit Configuration | Build and run" options to ensure that you are using the correct/consistent JDK versions. For this lab, it should be JDK 22 available in the PC.

Note that a "DiamondHunterEclipse.zip" is also provided in Moodle for you to download. Because this is an Eclipse project, you need to import and migrate it into your IntelliJ. The steps can be found here:
https://www.jetbrains.com/help/idea/import-project-from-eclipse-page-1.html

Now, let's have a closer look at the IntelliJ IDE "Navigate" menu.

1. ***How to find a specific class (open a class by name)?***
   Let's assume to find the class "GameState". An efficient way is to choose "Navigate/Class...", then in the "Classes" tab, search for "Game" and then double click the class "GameState" that is listed. This class will then appear in the editor window.

2. ***How to get an overview of methods and fields of a class?***
   First, put the cursor over the class name in the editor of the class's source code. Then choose "Navigate – File Structure". The methods and fields of the "GameState" class should all show up, as shown below.

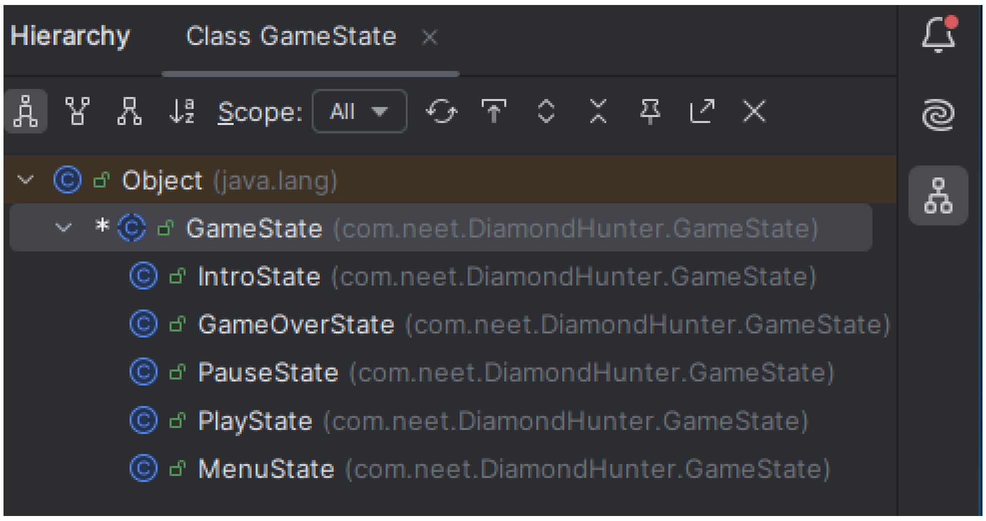


GameState.java

Inherited members (Ctrl+F12)   Anonymous Classes (Ctrl+I)   Lambdas (Ctrl+L)

```
GameState
    GameState(GameStateManager)
    draw(Graphics2D): void
    handleInput(): void
    init(): void
    update(): void
    gsm: GameStateManager
```

3. ***How to see the inheritance tree of a class?***
   First, put the cursor over and select the class name in the editor of the class's source code. Then choose "Navigate – Type Hierarchy". Then a panel will pop up on the right hand side showing the hierarchy tree of the class. For example, the GameState class is a subclass of the class Object, and a super class of the classes PlayState, IntroState, etc.



```
Object (java.lang)
  * GameState (com.neet.DiamondHunter.GameState)
      IntroState (com.neet.DiamondHunter.GameState)
      GameOverState (com.neet.DiamondHunter.GameState)
      PauseState (com.neet.DiamondHunter.GameState)
      PlayState (com.neet.DiamondHunter.GameState)
      MenuState (com.neet.DiamondHunter.GameState)
```

===
Task:
Try out some of the other options you find in the "Navigation" menu.
===

4. ***How can you find out in which classes a specific method (in this case, the update() method of the GameState class) is used?***
   You can select the method "update()", right click, the select "Find Usages". Two panels below should pop up telling you 2 usages are found. Both are in the GameSateManager source code, in lines 78 and 81.

The IntelliJ IDE provides many other little handy functions that make the life of a programmer easier. This includes, amongst others, support for refactoring code and for debugging and testing code. We will cover them in later lecture/laboratory exercises.