Exercise 1: Print Methods (2)

Prints out the name of the method, return type and parameter names and types

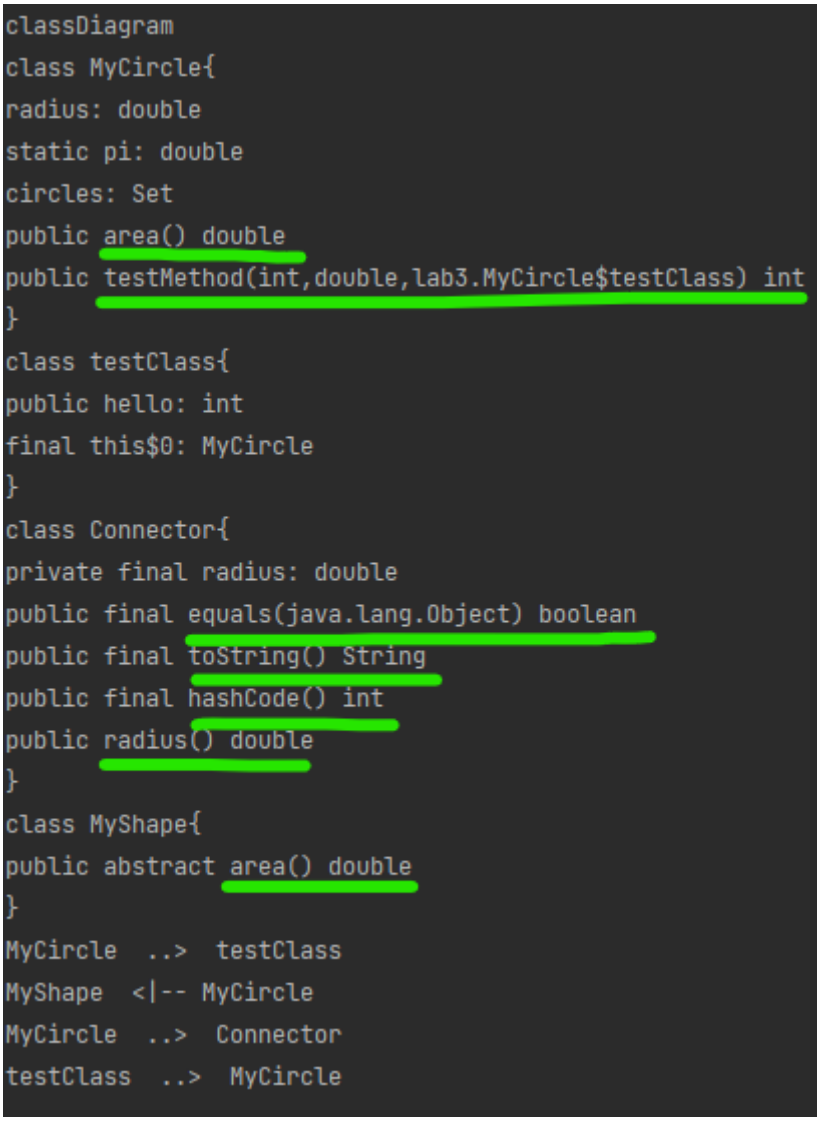
```
Inspecting class: lab3.MyCircle

Fields:
Name: radius : double
Name: pi : double
Name: circles : java.util.Set<java.util.Set<lab3.Connector>>

Constructors:
public lab3.MyCircle(double)
Name: lab3.MyCircle : double

public lab3.MyCircle(double,java.util.Set)
Name: lab3.MyCircle : double
Name: lab3.MyCircle : java.util.Set<java.util.Set<lab3.Connector>>

Methods:
public int lab3.MyCircle.testMethod(int,double,lab3.MyCircle$testClass)
Return Type: int
Argument type and name: (int arg0)
Argument type and name: (double arg1)
Argument type and name: (class lab3.MyCircle$testClass arg2)

public double lab3.MyCircle.area()
Return Type: double
```

Exercise 2: List Method Details (2)

Added Missing Methods along with its return, section responsible is below

```
System.out.println("Methods: ");
for (Method method : c.getDeclaredMethods()) {
    System.out.println(method);
    System.out.println("Return Type: " + method.getGenericReturnType());
    Parameter[] parameters = method.getParameters();
    for(Parameter p : parameters) {
        System.out.println("Argument type and name: (" + p.getType() + " " + p.getName() + ")");
    }
    System.out.println();
}
```

Output of the code above (ignoring for code for question 4)



Exercise 3: Fix problem with Parameterised Types (4)

This involves having an extra recursive function in ``findFields()`` where we check if it's a parameterized type, get the arguments for that type, check if the arguments contain classes that we have inputted into our system, then add it as a new link while also calling the recursive function again.

Find fields function implementation

```
public void findFields() {
    for (Class<?> c : classes) {
        for (Field f : c.getDeclaredFields()) {
            if (classes.contains(f.getType())) {
                links.add(new Link(c, f.getType(), LinkType.DEPENDANCY));
            }
            inspectType(c, f.getGenericType());
        }
    }
}
```

Recursive function implementation

```
private void inspectType(Class<?> c, Type type) {
    if(type instanceof ParameterizedType) {
        ParameterizedType t = (ParameterizedType) type;
        for(Type argument : t.getActualTypeArguments()) {
            if(classes.contains(argument)) {
                links.add(new Link(c,(Class<?>) argument, LinkType.DEPENDANCY));
            }
            inspectType(c, argument);
        }
    }
}
```

Dependency class implementation to take care of inner classes (findFields takes care of everything else)

```
public void findDependencies() {
    for (Class<?> c : classes) {
        for(Class<?> j : c.getDeclaredClasses())
        {
            links.add(new Link(c, j, LinkType.DEPENDANCY));
        }
    }
}
```

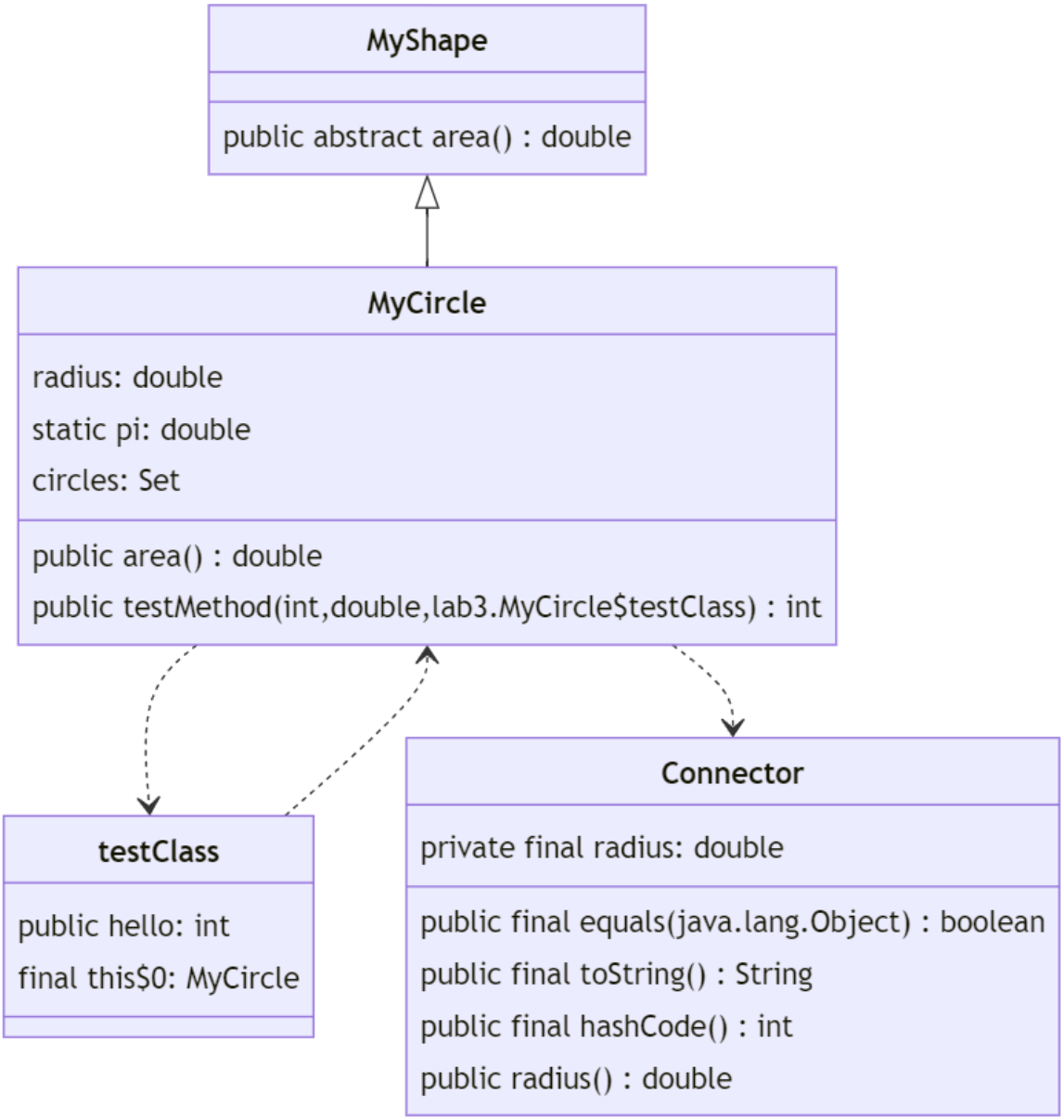
Code for output

```java
public class UMLShapesExample {
    public static void main(String[] args) {
        List<Class<?>> classes = new ArrayList<>();
        //add in all the classes we wish to generate UML for
        classes.add(MyShape.class);
        classes.add(MyCircle.class);
        classes.add(Connector.class);
        classes.add(MyCircle.testClass.class);
        ClassData cd = new ClassData(classes);
        System.out.println(cd.toMermaid());
    }
}
```

Output

```
classDiagram
class MyCircle{
radius: double
static pi: double
circles: Set
public area() double
public testMethod(int,double,lab3.MyCircle$testC
}
class testClass{
public hello: int
final this$0: MyCircle
}
class anotherTestClass{
public testing: int
final this$1: testClass
}
class MyShape{
public abstract area() dc
}
MyCircle  ..>  testClass
MyShape  <|-- MyCircle
MyCircle  ..>  Connector
testClass  ..>  MyCircle
```

Output in Mermaid Live



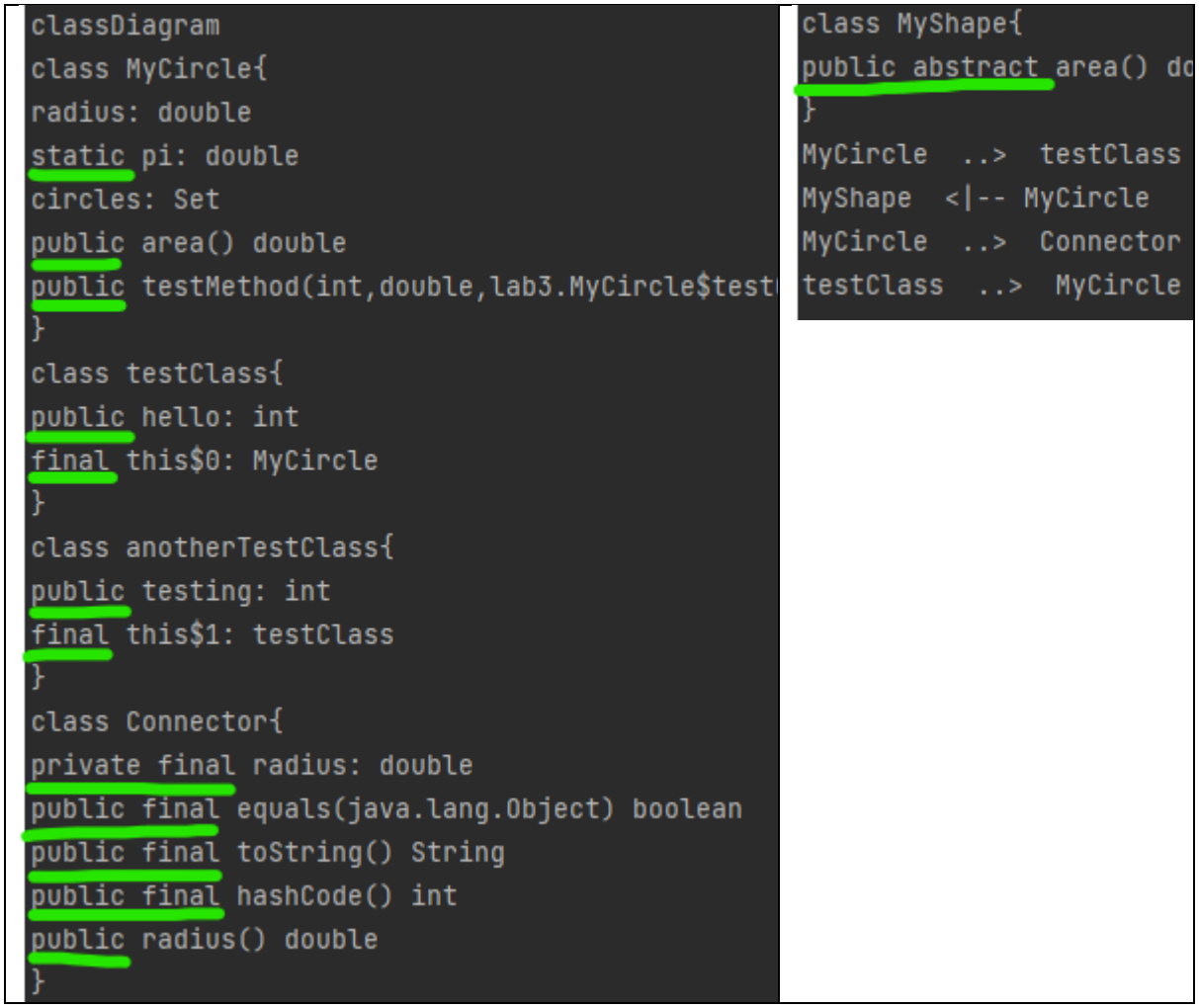
Exercise 4: Static Methods and Visibility Modifiers (2)

Added an extra function called append Modifiers which adds the modifiers to the beginning of each variable and method, if statement has 0 since 0 means that there are no modifiers on it

```
private static void appendModifiers(int modifier, StringBuilder sb) {
    if(modifier != 0)
    {
        sb.append(Modifier.toString(modifier) + " ");
    }
}
```

```java
public static String mermaidClassString(Class<?> c) {
    StringBuilder sb = new StringBuilder();
    sb.append("class ").append(c.getSimpleName()).append("{ \n");
    for (Field f : c.getDeclaredFields()) {
        appendModifiers(f.getModifiers(), sb);
        sb.append(f.getName())
                .append(": ")
                .append(f.getType().getSimpleName())
                .append("\n");
    }
    for(Method m : c.getDeclaredMethods())
    {
        appendModifiers(m.getModifiers(), sb);
        sb.append(m.getName())
                .append("(");
        Type[] parameterTypes = m.getGenericParameterTypes();
        for(Type type : parameterTypes) {
            sb.append(type.getTypeName() + ",");
        }
        if(parameterTypes.length > 0)
        {
            sb.deleteCharAt(sb.length() - 1);
        }
        sb.append(") ")
                .append(m.getReturnType().getSimpleName())
                .append("\n");
        for(Parameter p:m.getParameters()){
        }
    }
    sb.append("}\n");
    return sb.toString();
}
```

Output


classDiagram
class MyCircle{
radius: double
static pi: double
circles: Set
public area() double
public testMethod(int,double,lab3.MyCircle$test
}
class testClass{
public hello: int
final this$0: MyCircle
}
class anotherTestClass{
public testing: int
final this$1: testClass
}
class Connector{
private final radius: double
public final equals(java.lang.Object) boolean
public final toString() String
public final hashCode() int
public radius() double
}
class MyShape{
public abstract area() do
}
MyCircle ..> testClass
MyShape <|-- MyCircle
MyCircle ..> Connector
testClass ..> MyCircle