CS2030 Lecture 2

Testability in Object-Oriented Programming

Henry Chia (hchia@comp.nus.edu.sg)

Semester 2 2019 / 2020

Lecture Outline

- □ Testing classes using JShell
- □ Writing method tests as method chains
- □ Immutability
- □ Bottom-up testing of classes
- □ Factory methods
- □ Introduction to OOP principle of inheritance
    - – Super–sub (Parent–child) classes
    - – is-a relationship
    - – Overriding methods
- □ Cyclic dependency

Output a `Point` Object in JShell

- Thus far, creating an object using JShell results in the address of the object being output

```
jshell> new Point(1.0, 2.0)
$.. ==> Point@5c3bd550
```

- Make the output more meaningful by defining a `toString` method with the following method header:

```
class Point {
       ...
    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}
```

```
jshell> new Point(1.0, 2.0)
?.. ==> (1.0, 2.0)
```

- More details on `toString` when discussing method overriding

JShell as a “Testing Framework”

- We shall rely on JShell extensively when write tests
- Suppose we create a new `Point` and assign the reference to `p`:

```
jshell> Point p = new Point(1.0, 2.0)
p ==> (1.0, 2.0)

jshell> p.foo(..)
...
jshell> p.foo(..).bar(..)

jshell> p ???
```

- After a series of other commands involving method calls on `p`
  - What is the expected output of `p`?
  - Why make objects **immutable**?
  - Possible to design useful programs with immutable objects?

Mutators and its effect on Testing

□ Consider including mutators (or *setters*) in the `Point` class

```java
void setX(double x) {
    this.x = x;
}

void setY(double y) {
    this.y = y;
}
```

```
jshell> Point p = new Point(1.0, 2.0)
p ==> (1.0, 2.0)

jshell> p.setX(3.0)

jshell> p
p ==> (3.0, 2.0)
```

□ **`void`** methods that mutate state should be avoided

- Any mutation of state should be returned as a new object

Immutability

□ Methods should return new immutable objects

```
Point setX(double x) {
    return new Point(x, this.y);
}

Point setY(double y) {
    return new Point(y, this.x);
}
```

□ To prevent writing statements that violate immutability such as: `this.x = x`

– make all instance fields `final`

```
class Point {
    private final double x;
    private final double y;
```

Method Chaining

- Once an object is instantiated, it cannot be modified

```
jshell> Point p = new Point(1.0, 2.0)
$.. ==> (1.0, 2.0)
```

- Writing single-line tests on the object referenced by `p`
  - use method chaining

```
jshell> p.setX(3.0)
$.. ==> (3.0, 2.0)

jshell> p.setY(4.0)
$.. ==> (1.0, 4.0)

jshell> p.setX(5.0).setY(6.0)
$.. ==> (5.0, 6.0)
```

- Whichever way the tests are ordered, outcome is the same

Exercise: Moving a Point

- □ Define the method `moveBy` in class `Point` that moves the point located at (x, y) to the location $(x + dx, y + dy)$

```
Point moveBy(double dx, double dy) {
```

- □ Write some tests for the `moveBy` method

Bottom-up Testing

- With multiple classes, test bottom (standalone) class(es) first
- Having tested `Point`, continue “upwards” to test `Circle`

```
class Circle {
    private final Point centre;
    private final double radius;

    Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    boolean contains(Point point) {
        return centre.distanceTo(point) < radius;
    }

    String toString() {
        return "Circle centered at " + this.centre + " with radius " + radius;
    }
}
```

```
jshell> Circle c = new Circle(new Point(0.0, 0.0), 1.0)
c ==> Circle centered at (0.0, 0.0) with radius 1.0

jshell> c.contains(new Point(0.0, 0.0))
$.. ==> true

jshell> c.contains(new Point(1.0, 1.0))
$.. ==> false
```

Factory Methods

- What about the following test?

```
jshell> new Circle(new Point(0.0, 0.0), -1.0)
$.. ==> Circle centered at (0.0, 0.0) with radius -1.0
```

- To prevent the creation of invalid objects, **static** factory methods can be used to check the validity of the input parameters before generating the object

```
static Circle createCircle(Point centre, double radius) {
    if (radius > 0)
        return new Circle(centre, radius);
    else
        return null;
}
```

- Factory methods call the constructors to instantiate objects only if the parameters are valid, else a **null** value* is returned

* *Returning a **null** value is undesirable, but let's live with it for now..*

Factory Method

- Factory (or any **`static`**) methods are called via the class
- Constructors should now be made inaccessible to clients, i.e. need to make constructors **`private`**

```
jshell> new Circle(new Point(0.0, 0.0), -1.0)
|  Error:
|  Circle(Point,double) has private access in Circle
|  new Circle(new Point(0.0, 0.0), -1.0)
|  ^-----------------------------------^

jshell> Circle.createCircle(new Point(0.0, 0.0), 1.0)
$.. ==> Circle centered at (0.0, 0.0) with radius 1.0

jshell> Circle.createCircle(new Point(0.0, 0.0), -1.0)
$.. ==> null
```

`UnitCircle` as a Sub-Class of `Circle`

- □ Suppose we would like to represent another unit-circle object
  - What is the best way to design it? How about

```
static Circle createUnitCircle(Point centre) {
    return new Circle(centre, 1.0);
}
```

- □ Since a unit-circle is a type of circle, the **is-a** relationship is indicative of another OOP principle, namely **inheritance**
  - **is-a** relationship: `UnitCircle` is a `Circle`
  - `Circle` is the parent(super) class, while `UnitCircle` is the child(sub) class

```
class UnitCircle extends Circle {
    UnitCircle(Point centre) {
        super(centre, 1.0);
    }
}
```

Inheritance

□ Sub-class `UnitCircle` invokes the parent `Circle`'s constructor using `super(centre, radius)` within it's own constructor

- `Circle` constructor be made accessible from the sub-class
- Modify the accessibility of the constructor to **`protected`**

```
protected Circle(Point centre, double radius) {
    this.centre = centre;
    this.radius = radius;
}
```

□ If needed, a property of `Circle` (say `radius`) can also be made accessible to the child class by changing the access modifier

```
public class Circle {
    protected final double radius;
```

Inheritance

```
jshell> /open Point.java

jshell> /open Circle.java

jshell> /open UnitCircle.java

jshell> new UnitCircle(new Point(1.0, 1.0))
$.. ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> new UnitCircle(new Point(1.0, 1.0)).contains(new Point(1.0, 1.0))
$.. ==> true

jshell> new UnitCircle(new Point(1.0, 1.0)).contains(new Point(2.0, 2.0))
$.. ==> false
```

- Due to this is-a relationship, `Circle` methods can be invoked from `UnitCircle` objects

Instantiations of `Circle` *objects now possible, only because within the same package*

Overriding `toString` method

□ Invoking: `javadoc -d doc Circle.java`

```
public class Circle
extends java.lang.Object
...
public java.lang.String toString()

Returns a string representation of the Circle, showing its centre coordinates
and radius.

Overrides:
toString in class java.lang.Object

Returns:
a string representation of the Circle object.
```

□ This indicates that there is an equivalent `toString` method being overridden in the `java.lang.Object` class from which `Circle` extends (inherits)

Refer to https://www.comp.nus.edu.sg/ cs2030/javadoc/ for more details

Overriding `toString` Method

- All classes in Java inherit from the `Object` class
  - Methods defined in the `Object` class can be called from all objects of its child classes
- An example is the `toString` method
  - When JShell outputs the return value of an object created, it invokes the `toString` method
- Explicitly defining this `toString` method in our classes **overrides** the same method that is inherited from `Object`
  - The annotation `@Override` indicates to the compiler that the method overrides the same one in the parent class

Overriding `equals` Method

- Another commonly overridden method is the `equals` method
- Within the `Object` class, the `equals` method compares if two object references refer to the same object

```
jshell> new Point(0, 0) == new Point(0, 0)
$.. ==> false

jshell> new Point(0, 0).equals(new Point(0, 0))
$.. ==> false

jshell> new Point(0, 0).toString() == new Point(0, 0).toString()
$.. ==> false

jshell> new Point(0, 0).toString().equals(new Point(0, 0).toString())
$.. ==> true
```

- To have points with the same coordinate values deemed equal, we need to override the `equals` method inherited from `Object`

Overriding `equals` Method

- A naïve way of overriding the **`equals`** method is to define the method in the following way:

```
@Override
public boolean equals(Object obj) {
    Point p = (Point) obj;
    return Math.abs(this.x - p.x) < 1E-15 &&
        Math.abs(this.y - p.y) < 1E-15;
}
```

```
jshell> new Point(0,0).equals(new Point(0,0))
$.. ==> true
```

- Since the **`equals`** method takes in a parameter of **`Object`**
  - need to **type-cast** `obj` from **`Object`** type to **`Point`** type before accessing the radius in order to check for equality
- But what if the an object of different type is compared?
  - A ClassCastException is thrown

Overriding **`equals`** Method

- With a good sense of type awareness, the correct way to override the **`equals`** method is

```
@Override
boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (obj instanceof Point) {
        Point p = (Point) obj;
        return Math.abs(this.x - p.x) < 1E-15 &&
            Math.abs(this.y - p.y) < 1E-15;
    } else {
        return false;
    }
}
```

- In essence,
  - first check if it's the same object
  - then check if it's the same type
  - then check the associated equality property

Constructing Tests with **equals**

- Suppose there is a `midPoint` method

```
Point midPoint(Point otherPoint) {
    return new Point((this.x + otherPoint.x)/2,
                     (this.y + otherPoint.y)/2);
}
```

- Rather than "test" the actual output of the returned `Point` object via the `toString` method

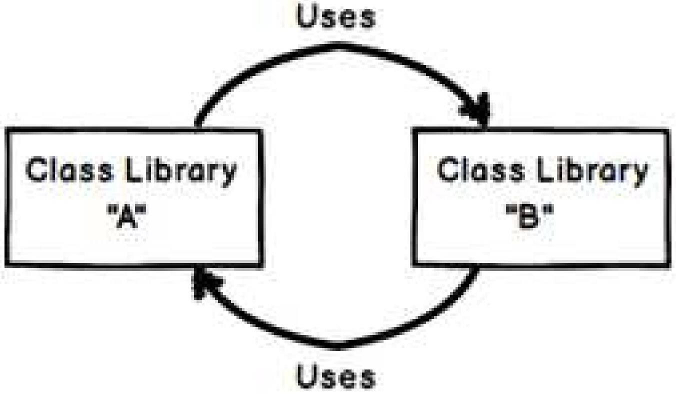
```
jshell> new Point(0, 0).midPoint(new Point(1, 1))
$.. ==> point (0.5, 0.5)
```

- The proper way is to test the equality between the actual `Point` object that is returned with the expected one

```
jshell> new Point(0, 0).midPoint(new Point(1, 1)).
   ...> equals(new Point(0.5, 0.5))
$.. ==> true
```

Cyclic Dependency

- □ Class dependency in the form of
  - *hard dependencies*: references to other classes in instance fields/variables
  - *soft dependencies*: references to other classes in methods (i.e. parameters, local variables, return type)
- □ Dependencies of classes/components **should not** have cycles
  - Avoid cyclic dependencies, e.g. testing class A requires class B to be tested first, and vice-versa



Cyclic Dependency

□ Using a simplified library system as an example, we would like to model the `Student` and `Book` class

```java
class Student {
    private final String name;
    private final Book book;

    Student(String name, Book book) {
        this.name = name;
        this.book = book;
    }

    String getName() {
        return this.name;
    }

    String getBookTitle() {
        return this.book.getTitle();
    }
}
```

```java
class Book {
    private final String title;
    private final Student student;

    Book(String title, Student student) {
        this.title = title;
        this.student = student;
    }

    String getTitle() {
        return this.title;
    }

    String getStudentName() {
        return this.student.getName();
    }
}
```

□ How do we set up a student to borrow a book?

□ How do we perform bottom-up testing?

Cyclic Dependency

- Use an association class to break the cyclic dependency
  - A student borrows a book under a **loan**

```java
class Student {
    private final String name;

    Student(String name) {
        this.name = name;
    }

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

class Book {
    private final String title;

    Book(String title) {
        this.title = title;
    }

    String getTitle() {
        return this.title;
    }
}
```

```java
class Loan {
    private final Student student;
    private final Book book;

    Loan(Student student, Book book) {
        this.student = student;
        this.book = book;
    }

    String getBookTitle() {
        return this.book.getTitle();
    }

    String getStudentName() {
        return this.student.getName();
    }
}
```

Lecture Summary

- □ Murphy's Law: *things that can go wrong, will go wrong*
- □ Objective of testing: *things that can go wrong, don't go wrong*
- □ The more flexible the software is, the more ways that things can go wrong, and the more tests are needed
- □ Appreciate that immutability decreases the flexibility of the software, leading to fewer tests
  - – Preventing internal state changes implies that there are no state transitions to test
- □ Appreciate why we need to break cyclic dependencies, so as to facilitate bottom-up testing
- □ Appreciate how to make software easier to test, maintain and more importantly, to reason