CS2030S Recitation

Week 4: Problem Set 2

brian

2025-09-04

National University of Singapore

Recap

About Dyamic binding

Why do we use method descriptor? Isn’t method signature not enough?

If you change the return type of a method in a superclass, you will get a runtime error if you do not correctly re-compile the subclass and just try to run things again

Recap: LSP

Liskov substitution principle

- Let $\varphi(x)$ be a property provable about objects x of type T. Then $\varphi(y)$ should be true for objects y of type S where $S <: T$
- Sounds like it contradicts polymorphism
  - Not really we only care about properties that we are interested in
  - They should still do the same thing, but maybe in a different way

Recap: Interfaces & Abstract classes

Abstract classes

- Sometimes we don’t have the full information to make a class
  - Humans all have 2 eyes, all have the same metabolic functions (methods)
  - But individual humans have different ways of talking, walking, etc
- Imagine writing an abstract `Human` class
  - For methods and fields we know, we just fill them in
  - For methods we don’t know how to implement yet, leave them abstract
  - classes that extend `Human` would have to implement the abstract methods

Recap: Interfaces & Abstract classes

Interfaces

- An interface has all methods abstract and *no* fields
  - These methods are always public
- It is a promise that anything that implements it has those methods
- A `LandVehicle` can be abstract with methods like `drive` etc
- Model what it can do (implementation left to implementations of the interface)

Recap: Casting an Interface

- You can almost always cast at compile time
    - Compiler can't prove that you might not have put a subclass that implements that interface so it allows it
    - Unless the CTT of whatever you are casting is `final`
    - If actually does not implement interface at run time then run time error

Problem Set

Q1 background

```java
public class Rectangle {
  private double width;
  private double height;
  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }
  public double getArea() {
    return this.width * this.height;
  }
  @Override
  public String toString() {
    return "Width: " + this.width + " Height: " + this.height;
  }
}
```

`Rectangle::getArea` is expected to return the product of its width and height

Q1a

We believe that `Square` inherits from `Rectangle`
`Square` instances must satisfy that the four sides are always the same length.

Create a class called `Square` with a single constructor method. Should have the following jshell output

```
jshell> new Square(5);
$.. ==> Width: 5.0 Height: 5.0

jshell> new Square(5).getArea();
$.. ==> 25.0
```

Q1a

```java
public class Square extends Rectangle {
 public Square(double length) {
   super(length, length);
 }
}
```

Q1b

Now Rectangle has two new methods to set the height and set the width.

```
public void setHeight(double height) {
  this.height = height;
}
public void setWidth(double width) {
  this.width = width;
}
```

Behaves like you would expect

```
jshell> Rectangle r = new Rectangle(5, 5);
jshell> r.setHeight(5);
jshell> r.setWidth(9);
jshell> r.getArea();
$.. ==> 45.0
```

Explain the undesirable effects for the Square class?

Q1b

- Explain the undesirable effects for the `Square` class?
  - `Square` inherits `setHeight` and `setWidth` methods from `Rectangle`
  - Height and width of the `Square` can now be independently set
  - The property of having all 4 sides being the same is no longer true

Q1c

Now we override these in Square

```java
@Override
public void setHeight(double height) {
    super.setHeight(height);
    super.setWidth(height);
}

@Override
public void setWidth(double width) {
    super.setHeight(width);
    super.setWidth(width);
}
```

Does this make sense? Should Square inherit from Rectangle?

Q1c

- Based on LSP, wherever we have a `Rectangle`, we should be able to put a `Square` and still have the same properties (that we choose)
- But if we put a `Square`, it would no longer adhere to the property of `Rectangle::getArea`
  - Example: Checking if a document is in landscape or portrait
- LSP is violated, so `Square` should not inherit from `Rectangle`

Q1d

Then should `Rectangle` inherit from `Square`?

- If `Rectangle` inherited from `Square`
- property of having all 4 sides the same is violated
- LSP is violated. `Square` and `Rectangle` should not inherit from each other

Q2a

We have the following code

```java
interface Shape{
  double getArea();
}
```

```java
interface Printable {
  void print();
}
```

```java
Circle c = new Circle(new Point(0, 0), 10);
Shape s = c;
Printable p = c;
```

Explain the compilation error (if any) in some of the statements below

(i) `s.print()`

(ii) `p.print()`

(iii) `s.getArea()`

(iv) `p.getArea()`

Q2a

- For i, `s` has compile time type `Shape` so does not have the `print` method. Thus a compilation error is thrown
- For ii, no error. `Printable` has `print` method and `p` is of compile time type `Printable`
- for iii, no error. `Shape` has `getArea` method and `s` is of compile time type `Shape`
- for iv, `p` has compile time type `Printable` so does not have the method `getArea`. Thus a compilation error is thrown

Q2b

Someone proposes to re-implement `Shape` and `Printable` as abstract classes instead. Would this work?

- No. Java does not allow the inheritance from multiple parent classes
- Abstract classes are still classes

Q2c

Can we define another interface `PrintableShape` which extends `Printable` and `Shape` and let `Circle` implement `PrintableShape` instead?

- Yes. Interfaces can inherit from multiple super-interfaces
- Note that the keyword is `extends` and not `implements` when interfaces inherit from other interfaces

Q2d

Give an example to illustrate why Java cannot inherit from multiple parent classes but can implement multiple interfaces.

- Say `A` has `foo()` which prints “a”
- `B` has `foo()` which prints “b”
- Let `C` inherit from both `A` and `B`, which `foo()` does it get?
- Interfaces are fine since if `A` and `B` are interfaces, then `C` is forced to implement `foo()`

The End

bye!