**CS2030S**

**RECITATION 3**

**Q1**

a)

i) No, as Shape does not implement Printable thus from the run-time type of a Shape object, it does not have the methods from a Printable interface

basically, no print() method in the shape interface. But if it is s.getArea(), then ok since it exists in the interface, which is overridden by the implementer’s getArea()

ii) Printable is an interface so it should be able to call methods ok

the print method in printable is also overridden by the implementer’s method

iii) calling getArea() from the runtime type of Shape, unless the getArea() has been overridden, should be fine? correct

interfaces force implementers to override the methods defined in the interfaces

iv)Printable does not have the method getArea() defined it it, so it will result in an error correct

b)No, as an object (in this case, Circle) can only extend from one class. Correct, since Java does not allow for multiple inheritance

c) assuming interfaces can extend each other, yes they could work

fun fact: interface can extends multiple interfaces and their methods ie PrintableShape extends Printable, Shape{} works.

interfaces allows polymorphism across different (non-related) classes by enforcing some methods unto implementers

dotted arrow (open triangle) implements

solid arrow (open triangle) => child of

**Q2**

There could be security implications as the child of multiple inheritance could access one of the parent class’s method whose intended purpose is in direct conflict with the other parent class’s intended purpose.

Eg one of the parent class A has a setter method while the other parent class B does not. Parent B’s omission of the setter method is such that the child classes cannot set their attributes but Parent A’s setter method enables the child class to set its attributes yea, I guess this is correct

**Eg class A has int foo(){return 1;}, B has int foo(){return 5;}**

**New C().foo() ==???????**

**Issue is that it leads to confusion due to ambiguity**

**Meanwhile, interfaces are ok because they do not have any body to define what the method is, but just directs that the implementer must override those methods.**

**Q3**

Yes, as the toggleUnderline() in PlainText should return an underlined Text (ie FormattedText object) yet when it returns this, it returns a PlainText object (ie no underlining) ok

Purpose of PlainText: not have any underlining at all

LSP: client of class T expects the same behaviour if T was substituted with a S <: T

If you switched the roles, the client would expect toggleUnderline does not change anything. But if you made formattedtext the child, when the client calls all of them, expecting toggleUnderline to do nothing, then FormattedText will underline the text. LSP still violated. To circumvent this, just remove toggleUnderline from the parent so client does not know toggleUnderline exists (looking from the parent’s perspective)

If the solution is to do a super.toggleUnderline, need to KIV the return type. But then it makes this redundant in the first place. So the solution is to just not provide the toggleUnderline method ;)

**Q4**

It complies. But if we swap the methods with the exact body, then it does not compile. Reason being B is a subclass of A, which means A cannot return B. But B can return an A object since it is the subclass of A.