CS2030 AY18/19 SEM 2

WEEK 3 | 1FEB 19

TA GAN CHIN YAO

ABOUT ME

- Year 2 Computer Science
- Took CS2030 2 sems back
- TA CS2030 last sem
- TA CS2030 this sem

ABOUT ME

- Taking only Tutorials, no Lab
- Teaching Tut 21/22/26
- Email me at gan@u.nus.edu

DISCLAIMER

Slides are made by me, unofficial, optional
Available to download at bit.ly/cs2030_gan
Slides (if any) will be uploaded on
Friday weekly

TUTORIAL

- Teach you core concepts
- Test your understanding
- Transferable to other languages

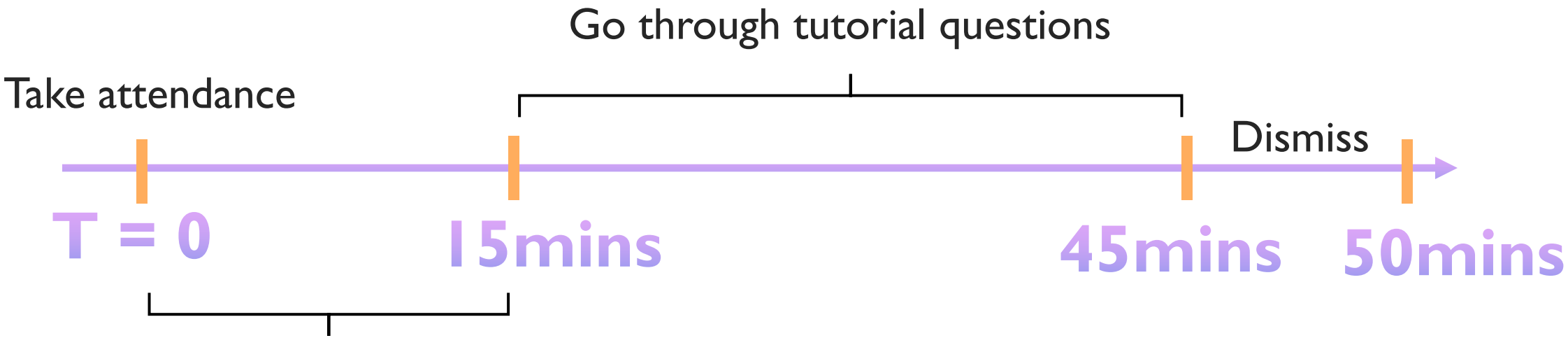
LAB

- Apply concepts
- See your concepts in action
- Practice coding for your future use

ABOUT CS2030 TUTORIAL

- 2% attendance, 3% class participation
- Try to attend your slot (space limited)
  If you cmi, you can attend any of my Tut 21/22/26
  Or any other TA slots (tell the TA to tell me, attendance track by individual TA)
- Class part:
  "90% would get 2 marks out of 3" – Prof. Henry

HOW I CONDUCT CS2030 TUTORIAL



Teach you core concepts
You have to tell me what you want me to talk about
Can be anything, e.g. What the heck is a class, explain polymorphism, what is inheritance T_T etc.

CONCEPTS

LEARN JAVA WELL

CS2030

CS2040

CS2103T

...

Q1.

Concepts:

Static variable VS instance variable

1. Given the following program fragment.

```
class A {
    public int x = 5;
    public static int y = 1;

    public A() {
        x = x + 1;
        y = y + 1;
    }
}
```

By either creating a `main` method or using `JShell`, invoke the following:

```
A a1 = new A();
A a2 = new A();
```

(a) After executing a1.x = 10, what is the value of a2.x?

6

Explanation: Since x is an instance variable, both a1 and a2 have their own copies of x. Therefore, assigning a1.x to a value of 10 will not change the value of a2.x

Q1.

1. Given the following program fragment.

```
class A {
    public int x = 5;
    public static int y = 1;

    public A() {
        x = x + 1;
        y = y + 1;
    }
}
```

By either creating a `main` method or using `JShell`, invoke the following:

```
A a1 = new A();
A a2 = new A();
```

(b) After executing a1.y = 10, what is the value of a2.y?

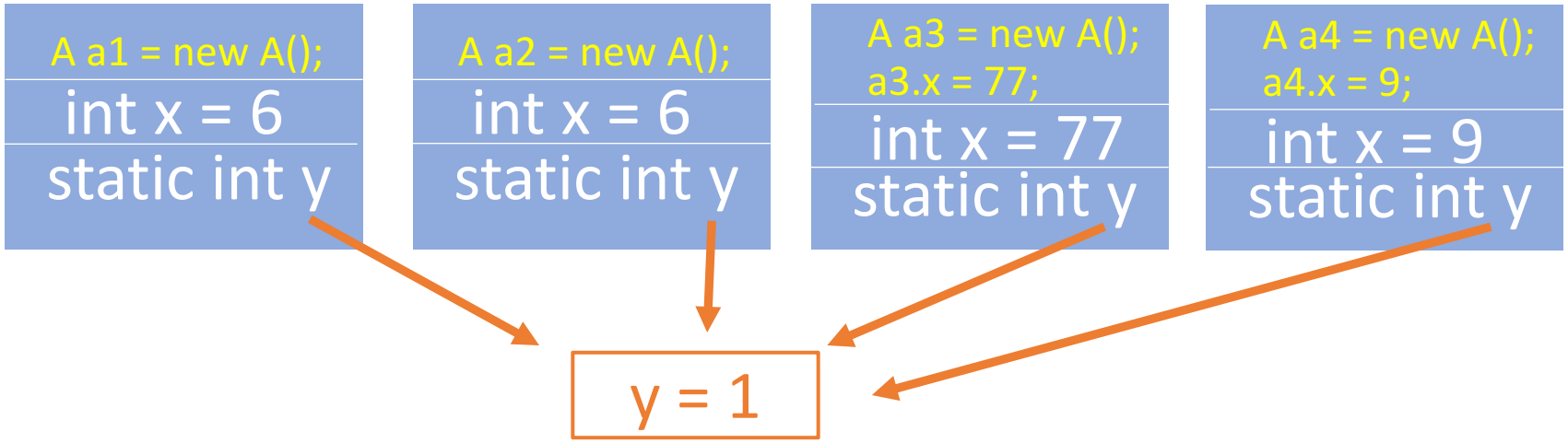
10

Explanation: Notice that y is a class variable by putting the keyword static. This means that only 1 copy of y is available for ALL instances of A. By changing the value of y via a1.y = 10, we are changing the 1 and only 1 copy of y. By accessing a2.y, we are accessing the same copy of y, which contains the value of 10. Even though a1.y = 10 is a valid statement, it is not a good practice to access a class variable via an object (a1 in this case). To make clear to reader that y is a class variable, we should access y via A.y = 10 instead.

Q1.

(c) What is the significance of the static keyword used during instance variable declaration? How is it useful?

"static" ensures that the variable belongs to the class. A static variable contains only 1 copy for any number of instances created. A static variable can be accessed without creating any object, as static variable is not associated with any object (instead it is associated with the class). We can access a static variable via *ClassName.StaticVariableName*. This is useful especially for creating a constant. Consider the constant *Math.PI*. It is declared as a static variable in *Math* java class so that we can access *PI* variable via the *Math* class name, instead of creating any *Math* object. Static variable is good if the variable should be associated with the class instead of any single object, i.e. the variable should contain the same value throughout for all instance objects created from the class.



Q1.

(d) Is A.x = 3 a valid statement? How about a1.x = 3 and A.y = 3?

- A.x = 3 is not a valid statement. You will get compile time error. This is because the syntax *ClassName.VariableOrMethodName* can only be apply to static variables or static method. Since x is not a static variable, we cannot use this notation.
- a1.x = 3 is perfectly valid. However, note that if x is a private variable, we cannot call a1.x = 3 outside of A class.
- A.y = 3 is the perfectly valid. In fact, this is the recommended way to access a static variable by using the notation *ClassName.StaticVariable*. Note that if y is a private static variable, we cannot access A.y outside of A class since y is now private.

Q2.

Concepts:
Access Modifier, Abstraction

2. Consider the following two classes:

```
public class P {
    private int x;
    public void changeSelf() {
        x = 1;   // ok
    }
    public void changeAnother(P p) {
        p.x = 1; // ok
    }
}
```

```
public class Q {
    public void changeAnother(P p) {
        p.x = 1;     // not ok
    }
}
```

(a) Which line(s) above violate the private access modifier of x?

p.x = 1; in class Q violates the private access modifier of x.

Q2.

Access modifier (modify access)

Most Restrictive ⟷ Least Restrictive

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Same rules apply for inner classes too, they are also treated as outer class properties

Q2.

2. Consider the following two classes:

```
public class P {
    private int x;
    public void changeSelf() {
        x = 1;
    }
    public void changeAnother(P p) {
        p.x = 1;
    }
}
```

```
public class Q {
    public void changeAnother(P p) {
        p.x = 1;
    }
}
```

(b) What does this say about the concept of an “abstraction barrier”?

- The abstraction barrier sits between the client and the implementer. Here class P is the implementer, and Q is the client that makes use of the p, an object of P.
- The barrier is not broken when one object of type P accesses the instance variables of another type P object, since P is the sole implementer.

Q3.

Concepts:

Method overriding, polymorphism

```
public class Circle {

    Point centre;
    double radius;

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

    @Override
    public boolean equals(Object obj) {
        System.out.println("equals(Object) called");
        if (obj == this) {
            return true;
        }
        if (obj instanceof Circle) {
            Circle circle = (Circle) obj;
            return (circle.centre.equals(centre) && circle.radius == radius);
        } else {
            return false;
        }
    }

    public boolean equals(Circle circle) {
        System.out.println("equals(Circle) called");
        return circle.centre.equals(centre) && circle.radius == radius;
    }
}
```

Q3.

```
Circle c1 = new Circle(new Point(0, 0), 10);
Circle c2 = new Circle(new Point(0, 0), 10);
Object o1 = c1;
Object o2 = c2;
```

What is the output of the following statements?

(a) o1.equals(o2);

- equals(Object) called
- => false

(b) o1.equals((Circle) o2);

- equals(Object) called
- => false

(c) o1.equals(c2);

- equals(Object) called
- ==> false

(d) o1.equals(c1);

- equals(Object) called
- ==> true

Q3.

```
Circle c1 = new Circle(new Point(0, 0), 10);
Circle c2 = new Circle(new Point(0, 0), 10);
Object o1 = c1;
Object o2 = c2;
```

What is the output of the following statements?

(e) c1.equals(o2);

- equals(Object) called
- ==> false

(f) c1.equals((Circle) o2);

- equals(Circle) called
- ==> false

(g) c1.equals(c2);

- equals(Circle) called
- ==> false

(h) c1.equals(o1);

- equals(Object) called
- ==> true

Q3.

- Note: Observe that in Circle class, it overrides the **equals(Object obj)** method in Object class, and overload it with another **equals(Circle circle)** method. Understand that for the line *Object obj = new Circle(...),* obj is of type *Object*, but points to a *Circle* object. (Recall that superclass reference can point to subclass object, but not the other way round).
- Calling the *equals* method though a reference of type *Object* would invoke the *toString* method of Object, but which is overridden by the same method of the sub-class Circle.
- The only time that the overloaded method **equals(Circle circle)** can be called is when the method is invoked through an object of *Circle* type, and the argument is an object of *Circle* type also.
- TLDR: Object class has no knowledge of **equals(Circle circle)**. For overridden method to work, your superclass must contains that method signature.

Q4.

4. Which of the following program fragments will result in a compilation error?

Concepts:
Method overloading, method signature

(a)
```
class A {
    public void f(int x) {}
    public void f(boolean y) {}
}
```

- Compiles fine.

(b)
```
class A {
    public void f(int x) {}
    public void f(int y) {}
}
```

- Compile error. Both methods f have the same **method signature**. The variable name in the parameter does not differentiate the different methods. Method signature refers to the **name** of the method, the **parameter type** of the method, the **number** of parameters, and the **order** of the parameters. For **method overloading** to work, the methods need to have **different method signature**.

Q4.

4. Which of the following program fragments will result in a compilation error?

(c)
```
class A {
    private void f(int x) {}
    public void f(int y) {}
}
```

- Compile error. Both methods f have the same method signature. Access modifier (public private..) is **not** part of method signature.

(d)
```
class A {
    public int f(int x) {
        return x;
    }
    public void f(int y) {}
}
```

- Compile error. Both methods f have the same method signature. Return type is **not** part of method signature, as the Java compiler cannot differentiate methods solely on return type.
- Why **return type** is not part of **method signature**?
  Ans: https://stackoverflow.com/questions/13314316/why-is-the-return-type-of-method-not-included-in-the-method-signature

Q4.

4. Which of the following program fragments will result in a compilation error?

(e)
```
class A {
    public void f(int x, String s) {}
    public void f(String s, int y) {}
}
```

- Compiles fine. Both methods f have different method signature.
- The order of parameter is important.

Q5.

Concepts:
Inheritance, Liskov Substituition Principle

5. In Lecture #3, we designed the class `Rectangle` that inherits from the class `Shape`. Now we want to design a class `Square` that inherits from `Rectangle`. A square has the constraint that the four sides are of the same length.

   (a) How should `Square` be implemented such that we obtain the following using `JShell`?

```
jshell Shape.java Rectangle.java Square.
java
|  Welcome to JShell -- Version 9.0.4
|  For an introduction type: /help intro

jshell> Square s = new Square(5)
s ==> Square with area 25.00 and perimeter 20.00

jshell>
```

- Override the ***toString()*** method in Square class to print out the necessary output.
- Write a method to calculate the area of a square (length * length), and the parameter of the square (length * 4).

Q5.

(b) Do you think `Square` should inherit from `Rectangle`? Or should it be the other way around? Or maybe they should not inherit from each other?

- They should not inherit from each other at all.
- Suppose Square inherits from Rectangle, so methods in Rectangle gets inherited by Square.
- One method that can be included into class Rectangle is the resize(int width, int height) method that resizes the rectangle object.
- Now, suppose a client of the rectangle class receives Rectangle objects (some of which could be of type Square). By applying resize on these objects, one can turn a square into a rectangle since the method resize is inherited from Rectangle to a Square. Even if setSize can be overridden in Square, which of the two parameters, height or width would it use?
- Realise that LSP will be violated in this case. The essence of inheritance is this: Whenever an object of a superclass is expected, one can use an object of subclass for the purpose. This is because an object of type subclass is **ALSO** an object of type superclass. There SHOULD NOT be any situation whereby a subclass object cannot be use in place of a superclass object. If this happens, then this is a strong indicator that there should not be an inheritance relationship.
- Tip: A “special case” should most often not be made into an inheritance relationship. E.g. a Square is a special case of a rectangle. Ensure that subclass is completely a superclass object in order to form a good inheritance relationship. E.g. a dog is completely an animal. Therefore, Dog extends Animal is a reasonable and good relationship.

SUMMARY CONCEPTS

- Static variable
- Instance variable
- Access modifier
- Abstraction
- Method overriding
- Polymorphism
- Method overloading
- Method signature
- Inheritance

If you wish to read in more detailed:
https://nus-cs2030.github.io/1718-s2/lec01/index.html

QUESTIONS?