CS2030 AY18/19 SEM 2

WEEK 5 | 15 FEB 19

TA GAN CHIN YAO

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

CONCEPTS

Q1.
a.

Concepts:
Interface, interface type referencing actual object

1. Given the following interfaces.

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

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

(a) Suppose class `Circle` implements both interfaces above. Given the following program fragment,

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

Are the following statements allowed? Why do you think Java does not allow some of the following statements?

i.e.
public class Circle implements Shape, Printable {}

i) s.print();

Not allowed. Compile error, print() not found

Explanation: As far as the compiler is concern, s is declared as type Shape. No matter the object s is actually pointing to, the first step is to go to Shape interface and see if the method exist. In this case, print() does not exist in Shape interface, hence cannot be found.

Note that Shape s = new Circle() is similar in concepts to superclass variable referencing subclass object. Java does not allow s.print() because even though it is possible that some class such as Circle in this case implements Printable and actually have s points to a Circle object (thus seemingly makes print() visible via the Circle), it is also entirely possible that a class only implements the Shape interface and referenced by Shape s, e.g. public class Circle implements Shape {} Shape s = new Circle(); The general rule in Java is that if anything is ambiguous, it will not compile.

Q1.
a.

ii) p.print();

Allowed, perfectly valid.

Explanation:
p is declared as type Printable. When we call p.print(), compiler will see the type of p, which is Printable. Then, it will go to interface Printable and see if print() exists. In this case, it does. No matter what p is pointing to, it **must** have implemented Printable.

Imagine if Circle did not implement Printable, and we call Printable p = new Circle();
You will get a compile error straight away saying "error: incompatible types: Circle cannot be converted to Printable".
Hence, before we even reach the line p.print(), the line Printable p = new XXX() must succeed. In order for it to succeed, the object being referenced by p must implement Printable (or its sub interface). And we can be sure that p is referring to something that implements Printable itself. Then, polymorphism will play and the overriden print() in Circle will be called.

iii) s.getArea();

Allowed, perfectly valid.

Explanation:
Similar explaination to part (ii).

s is declared as type Shape, and getArea() is a method that exists in the Shape interface. Hence, we can be guaranteed that s is referring to an object that implements type Shape.

Q1.
a.

iv) p.getArea();

Not allowed. Compile error, getArea() not found

Explanation:
Similar explaination to part (i).

p is declared as type Printable. When we call p.getArea(), compiler will first look up Printable interface and check if getArea() exists. It does not exist, hence compile error is thrown.

The idea is that Java does not want to allow ambiguity. While in this case it is clear cut that p is pointing to a Circle object that actually implements Shape and implements the getArea() method, it could be perfectly valid for Circle to only implements Printable, and not Shape. Then, Printable p = new Circle() will still work, but now getArea() method is missing from Circle.

Hence to prevent ambiguity, Java adopts the principle that if it is 100% safe and possible, it will compile. If there is a chance that an ambiguity occurs, it does not allow you to compile.

Q1.
b.

Concepts:
Inheritance

(b) Someone proposes to re-implement Shape and Printable as abstract classes instead? Would this work?

No. You cannot inherit from multiple parent classes.

```
// OK
abstract class Shape {
    public abstract double getArea();
}

// OK
abstract class Printable {
    public abstract void print();
}

// NOT OK, cannot inherit more than 1 class
public class Circle extends Shape, Printable {

}
```

Q1.

c.

(c) Can we define another interface `PrintableShape` as

```
public interface PrintableShape extends Printable, Shape {
}
```

and let class `Circle` implement `PrintableShape` instead?

Yes, it is allowed. Interfaces can inherit from multiple parent interfaces.

Take note that here we are using the concept of extends, not implements.
We implements an interface from a class. An interface can have inheritance relationship just like normal class. When one interface is inheriting from another interface, it is using extends keyword.

An **interface** can **extend multiple interfaces**.
A **class** can **implement multiple interfaces**.
However, a **class** can only **extend a single class (no matter abstract/concrete)**.

Q1.
c.

Side note:
Why can interface extends from multiple interfaces, but class cannot extend from multiple classes?

Simply put, if a class can extends from multiple classes, it will cause complicated problem such as the "Diamond problem":
https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

Java designers do not want multiple inheritance from classes as they think that most (if not all) problems could be solve without multiple inheritance through the use of interfaces. They also want to prevent inheritance abuse whereby people simply keep inheriting from other classes where there could be better way of design (such as through composition instead).

Java designers feel that the benefit of multiple inheritance is lesser than the problem multiple inheritance can bring. Hence, they don't allow multiple inheritance when designing Java.

Q2.

Concepts:

Interface, using interface

2. Write a class `Rectangle` that implements the two interfaces in question 1. You should make use of two diagonally-opposite points (bottom-left and top-right) to define the rectangle. How do you handle the case that the two points do not define a proper rectangle?

   Assume that the sides of the rectangles are parallel with the x- and y-axes (in other words, the sides are either horizontal or vertical).

Q2.

```
public class Rectangle implements Shape, Printable {
    Point bottomLeft;
    Point topRight;

    private Rectangle(Point bottomLeft, Point topRight) {
            this.bottomLeft = bottomLeft;
            this.topRight = topRight;
    }

    public static Rectangle getRectangle(Point bottomLeft, Point topRigth) {
        if (getLength(bottomLeft, topRight) > 0 &&
                getHeight(bottomLeft, topRight) > 0) {
            return new Rectangel(bottomLeft, topRight);
        } else {
            return null;
        }
    }

    private static double getLength(Point bottomLeft, Point topRight) {
        return topRight.getX() - bottomLeft.getX();
    }

    private static double getHeight(Point bottomLeft, Point topRight) {
        return topRight.getY() - bottomLeft.getY();
    }

    private double getLength() {
        return getLength(this.bottomLeft, this.topRight);
    }

    private double getHeight() {
        return getHeight(this.bottomLeft, this.topRight);
    }

    public double getArea() {
        return getLength() * getHeight();
    }

    public void print() {
        System.out.println("Printable...");
    }
}
```

← This is just one way of doing. You can have other ways as well. Here we need a method getRectangle() to return null. We cannot return null in a constructor. Hence, we private the constructor to prevent people from calling, and the intended way of creating Rectangle is through the static method getRectangle()

Possible ways of handling the case where two points do not define a proper rectangle:

```
boolean notProperRectangle;

// Way 1, to return null. States properly inside your documentation
// that you are returning null for other developer to know.
if (notProperRectangle) {
    return null;
}

// Way 2, slight enhancement from way 1
if (notProperRectangle) {
    System.out.println("Your Rectangle is not valid");
    // can proceed one step further to say why not valid,
    // i.e. maybe bottomLeft is on right of topRight
    return null;
}

// Way 3, recommended. Because sometimes when you return a null
// Rectangle, it might cause further problem down the road where
// developers end up working with a null Rectangle and get
// nullPointerException
if (notProperRectangle) {
    throw new IllegalArgumentException("Your Rectangle is not valid");
}
```

Q2.

```
public class Rectangle implements Shape, Printable {
    Point bottomLeft;
    Point topRight;

    public Rectangle(Point bottomLeft, Point topRight) {
        if (getLength(bottomLeft, topRight) > 0 &&
                getHeight(bottomLeft, topRight) > 0) {
            this.bottomLeft = bottomLeft;
            this.topRight = topRight;
        } else {
            throw new IllegalArgumentException(bottomLeft + " " +
                    topRight);
        }
    }

    private static double getLength(Point bottomLeft, Point topRight) {
        return topRight.getX() - bottomLeft.getX();
    }

    private static double getHeight(Point bottomLeft, Point topRight) {
        return topRight.getY() - bottomLeft.getY();
    }

    private double getLength() {
        return getLength(this.bottomLeft, this.topRight);
    }

    private double getHeight() {
        return getHeight(this.bottomLeft, this.topRight);
    }

    public double getArea() {
        return getLength() * getHeight();
    }

    public void print() {
        System.out.println("Printable...");
    }
}
```

← Another possible solution. Notice here that we can explicitly declare a Rectangle constructor since we are not returning null

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextDouble()) {
            Point bottomLeft = new Point(sc.nextDouble(), sc.nextDouble());
            Point topRight = new Point(sc.nextDouble(), sc.nextDouble());
            try {
                Rectangle r = new Rectangle(bottomLeft, topRight);
                System.out.println(r.getArea());
            } catch (IllegalArgumentException ex) {
                System.err.println(ex);
            }
        }
    }
}
```

Q3.

Concepts:

Interface, constructor chaining

3. Let's now extend our shapes from two-dimensional to three dimensional.

   (a) Write an interface called `Shape3D` that supports a method `getVolume`. Write a class called `Cuboid` that implements `Shape3D` and has three private `double` fields `length`, `height`, and `breadth`. The method `getVolume()` should return the volume of the `Cuboid` object. The constructor for `Cuboid` should allow the client to create a `Cuboid` object by specifying the three fields length, height and breadth.

   (b) Write a new interface `Solid3D` that inherits from interface `Shape3D` that supports two methods: `getDensity()` and `getMass()`.

   (c) Now, write a new class called `SolidCuboid` with an additional private `double` field `density`. The implementation of `getDensity()` should return this field while `getMass()` should return the mass of the cuboid. The `SolidCuboid` should call the constructor of `Cuboid` via `super` and provides two constructors: one constructor that allows the client to specify the density, while the other does not and just sets the default density to 1.0.

   (d) Test your implementation with by writing a suitable client class.

Q3.

```
public interface Shape3D {
    double getVolume();
}

public interface Solid3D extends Shape3D {
    double getDensity();
    double getMass();
}

public class Cuboid implements Shape3D {
    private double length;
    private double height;
    private double breadth;

    public Cuboid(double length, double breadth, double height) {
        this.length = length;
        this.breadth = breadth;
        this.height = height;
    }

    public double getVolume() {
        return length * height * breadth;
    }
}

public class SolidCuboid extends Cuboid implements Solid3D {
    private double density;

    public SolidCuboid(double length, double height, double breadth,
            double density) {
        super(length, height, breadth);
        this.density = density;
    }

    public SolidCuboid(double length, double height, double breadth) {
        this(length, breadth, height, 1.0);
    }

    public double getDensity() {
        return density;
    }

    public double getMass() {
        return getVolume() * density;
    }
}
```

Note that by default, methods in interface are implicitly public abstract, so you can omit these 2 keywords

Better to annotate @Override for overriden method
https://stackoverflow.com/questions/94361/when-do-you-use-javas-override-annotation-and-why

Calling this constructor

Calling this constructor

Q4.

Concepts:

Inheritance

4. Write each of the following program fragments using `jshell`. Will it result in a compilation or runtime error? If not, what is the output?

(a)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
}

B b = new B();
b.f();
A a = b;
a.f();
```

Compiles fine.
Output:
A f
A f

b is declared as type B. Goes to class B, did not find method f(). Goes to its superclass A, found f() there. Hence, call f() in class A

Though class B does not contain f(), methods in superclass are inherited by subclass

Q4.

Concepts:

Polymorphism

(b)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        System.out.println("B f");
    }
}

B b = new B();
b.f();
A a = b;
a.f();
a = new A();
a.f();
```

Method overriding

Compiles fine.
Output:
B f
B f
A f

b is declared as type B. Goes to class B, found method f(). b is pointing to an object of type B. Hence, call f() in class B

Polymorphism at play.
a is declared as type A. Goes to class A, found method f(). Looks at what a is pointing to. a is pointing to an object of type B. Goes to B and found overriden method f(). Calls f() in class B

a is declared as type A. Goes to class A, found method f(). a is pointing to an object of type A. Hence, call f() in class A

Q4.

Concepts:

Polymorphism, 'super' keyword

(c)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        super.f();
        System.out.println("B f");
    }
}

B b = new B();
b.f();
A a = b;
a.f();
```

Calling

Compiles fine.
Output:
A f
B f
A f
B f

Polymorphism at play.
Calls f() in class B

Q4.

(d)

```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        this.f();
        System.out.println("B f");
    }
}

B b = new B();
b.f();
A a = b;
a.f();
```

Compiles fine.
But results in infinite recursion when executed.

Calling f() again, resulting in infinite recursion

Infinite recursion, eventually resulting in stack overflow

Polymorphism. Call f() in class B. Note that the keyword this in this.f() refers to the object itself, in this case referring to an object of type B. It will call f() in class B again

Q4.

Concepts:
Method signature, override

(e)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    int f() {
        System.out.println("B f");
        return 0;
    }
}

B b = new B();
b.f();
A a = b;
a.f();
```

Compile error.

```
Error:
f() in B cannot override f() in A
  return type int is not compatible with void
    int f() {
    ^--------...
```

Same method name f() as its superclass A. But method signature is different. Hence, cannot override, and you have 2 methods with same name f(). Therefore, compile error.

Q4.

(f)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f(int x) {
        System.out.println("B f");
    }
}

B b = new B();
b.f();
b.f(0); // Ok. Goes to class B and found f(int x)
A a = b;
a.f();
a.f(0);
```

Compile error.

```
|  Error:
|  method f in class A cannot be applied to given types;
|    required: no arguments
|    found: int
|    reason: actual and formal argument lists differ in length
|  a.f(0);
|  ^-^
```

Ok. Goes to class B, cannot find f(). Goes to superclass A. Found f(). Therefore, call f() in class A

Ok. a is declared as type A. Goes to class A and found f(). Since f() is not overriden in subclass, it will call f() in class A

Compiles error. a is declared as type A. Goes to class A and cannot find f(parameter takes in 0). Goes to all superclasses of A, also cannot find f(parameter takes in 0). Therefore compile error.

Q4.

Concepts:
Polymorphism

(g)
```
class A {
    public void f() {
        System.out.println("A f");
    }
}

class B extends A {
    public void f() {
        System.out.println("B f");
    }
}

B b = new B();
A a = b;
a.f();  // Polymorphism at play. Calls f() in class B
b.f();
```

Compiles fine.
Output:
B f
B f

Q4.

Concepts:

Access modifier

(h)
```
class A {
    private void f() {
        System.out.println("A f");
    }
}

class B extends A {
    public void f() {
        System.out.println("B f");
    }
}

class Main {
    public static void main(String[] args) {
        B b = new B();
        A a = b;
        a.f();
        b.f();

    }
}
```

Compile error

a is declared as type A. Goes to class A. f() is private in A. Therefore, compile error, cannot call private from outside the class

Ok. Output "B f"

Q4.

Concepts:
static methods

(i)
```
class A {
    static void f() {
        System.out.println("A f");
    }
}

class B extends A {
    public void f() {
        System.out.println("B f");
    }
}

B b = new B();
A a = b;
a.f();
b.f();
```

Compile error
Cannot override static method.

Why can't we override static method?
https://stackoverflow.com/questions/2223386/why-doesnt-java-allow-overriding-of-static-methods

Q4.

(j)
```
class A {
    static void f() {
        System.out.println("A f");
    }
}

class B extends A {
    static void f() {
        System.out.println("B f");
    }
}

B b = new B();
A a = b;
A.f(); // Output: A f
B.f(); // Output: B f
a.f(); // Output: A f. Since static method is not overriden, there is no polymorphism
b.f(); // Output: B f.
```

Compiles fine.
Output:
A f
B f
A f
B f

Here 2 static methods do not override.
When you write a new static method in subclass with the same method signature, the old static method in superclass is hidden, not overriden
https://stackoverflow.com/questions/10291949/are-static-methods-inherited-in-java

Q4.

Concepts:
Access modifier

```
(k)  class A {
         private int x = 0;
     }

     class B extends A {
         public void f() {
             System.out.println(x);
         }
     }

     B b = new B();
     b.f();
```

Compile error
x is private in superclass. Cannot access private field outside of its class, not even from subclass.

https://stackoverflow.com/questions/4716040/do-subclasses-inherit-private-fields

Q4.

Concepts:
Access modifier, 'super' keyword

(l)
```
class A {
    private int x = 0;
}

class B extends A {
    public void f() {
        System.out.println(super.x);
    }
}

B b = new B();
b.f();
```

Compile error
x is private in superclass. Cannot access private field outside of its class, not even from subclass. Not even with super.x

Q4.

Concepts:

Access modifier

(m)
```
class A {
    protected int x = 0;
}

class B extends A {
    public void f() {
        System.out.println(x);
    }
}

B b = new B();
b.f();
```

Compiles fine.

Output:

0

Ok. Protected variable can be accessed from subclass

Q4.

Concepts:

Variable shadowing

(n)
```
class A {
    protected int x = 0;
}

class B extends A {
    public int x = 1;
    public void f() {
        System.out.println(x);
    }
}

B b = new B();
b.f();
```

Compiles fine.
Output:
1

This x shadows the x in superclass, even if access modifier is different. Hence, anywhere in this class when we call x, this x = 1 will be called. If we want to refer to superclass x, we need to call super.x

Q4.

(o)
```
class A {
    protected int x = 0;
}

class B extends A {
    public int x = 1;
    public void f() {
        System.out.println(super.x);
    }
}

B b = new B();
b.f();
```

Compiles fine.
Output:
0

Calling x in superclass A.
Note: if x is private in A, then super.x will cause compile error. If there is no x in A, then super.x will also cause compile error even if there is x in B.

SUMMARY CONCEPTS

- Interface
- Constructor chaining
- Inheritance
- Polymorphism
- ‘super’ keyword
- Method signature
- Method override
- Access modifier
- Static methods
- Variable shadowing

IMPORTANT

Every class in java extends from Object class implicitly.
i.e. public class Circle {} is actually public class Circle extends Object {}
That means every class you create can use the following methods:

Constructor Summary

Constructors

Constructor	Description
`Object()`	Constructs a new object.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods** | **Deprecated Methods**

Modifier and Type	Method	Description
protected `Object`	`clone()`	Creates and returns a copy of this object.
boolean	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
protected void	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
int	`hashCode()`	Returns a hash code value for the object.
void	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
void	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.
void	`wait()`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
void	`wait(long timeout)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
void	`wait(long timeout, int nanos)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

Java Object class API SE10

In CS2030, we are only interested in in equals(obj), toString() and hashCode()

```java
/**
 * Indicates whether some other object is "equal to" this one.
 * <p>
 * The {@code equals} method implements an equivalence relation
 * on non-null object references:
 * <ul>
 * <li>It is <i>reflexive</i>: for any non-null reference value
 *     {@code x}, {@code x.equals(x)} should return
 *     {@code true}.
 * <li>It is <i>symmetric</i>: for any non-null reference values
 *     {@code x} and {@code y}, {@code x.equals(y)}
 *     should return {@code true} if and only if
 *     {@code y.equals(x)} returns {@code true}.
 * <li>It is <i>transitive</i>: for any non-null reference values
 *     {@code x}, {@code y}, and {@code z}, if
 *     {@code x.equals(y)} returns {@code true} and
 *     {@code y.equals(z)} returns {@code true}, then
 *     {@code x.equals(z)} should return {@code true}.
 * <li>It is <i>consistent</i>: for any non-null reference values
 *     {@code x} and {@code y}, multiple invocations of
 *     {@code x.equals(y)} consistently return {@code true}
 *     or consistently return {@code false}, provided no
 *     information used in {@code equals} comparisons on the
 *     objects is modified.
 * <li>For any non-null reference value {@code x},
 *     {@code x.equals(null)} should return {@code false}.
 * </ul>
 * <p>
 * The {@code equals} method for class {@code Object} implements
 * the most discriminating possible equivalence relation on objects;
 * that is, for any non-null reference values {@code x} and
 * {@code y}, this method returns {@code true} if and only
 * if {@code x} and {@code y} refer to the same object
 * ({@code x == y} has the value {@code true}).
 * <p>
 * Note that it is generally necessary to override the {@code hashCode}
 * method whenever this method is overridden, so as to maintain the
 * general contract for the {@code hashCode} method, which states
 * that equal objects must have equal hash codes.
 *
 * @param   obj   the reference object with which to compare.
 * @return  {@code true} if this object is the same as the obj
 *          argument; {@code false} otherwise.
 * @see     #hashCode()
 * @see     java.util.HashMap
 */
public boolean equals(Object obj) {
    return (this == obj);
}
```

equals(Object obj) source code, jdk10 Object class

```java
/**
 * Returns a string representation of the object. In general, the
 * {@code toString} method returns a string that
 * "textually represents" this object. The result should
 * be a concise but informative representation that is easy for a
 * person to read.
 * It is recommended that all subclasses override this method.
 * <p>
 * The {@code toString} method for class {@code Object}
 * returns a string consisting of the name of the class of which the
 * object is an instance, the at-sign character `{@code @}', and
 * the unsigned hexadecimal representation of the hash code of the
 * object. In other words, this method returns a string equal to the
 * value of:
 * <blockquote>
 * <pre>
 * getClass().getName() + '@' + Integer.toHexString(hashCode())
 * </pre></blockquote>
 *
 * @return  a string representation of the object.
 */
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

toString() source code, jdk10 Object class

Interface

- All methods are public, abstract by default
- When overriding interface method, must declare public as the access modifier
- May have variables only if they are final, static, and initialized
- No instance variable allowed
- Class can implement multiple interfaces
- Interface can extend one or more other interfaces
- You can give method body to an interface by specifying 'default' keyword (Java 8 onwards)
- default methods do not need to be overriden in implementing class
- static method must have a body (cannot be abstract)
- static methods are not inherited
- Cannot instantiate interface object,
  i.e. no new Interface();

Abstract

- Abstract class is also a class (all behaviour similar to normal class, except instantiation)
- A class containing at least one abstract method must be an abstract class
- All abstract class must be overriden in subclass unless the subclass is abstract as well
- Normally we call non-abstract class concrete class
- Cannot instantiate an abstract class
- Cannot combine final abstract class together because abstract class is meant to be inherited
- Can have abstract methods, and non-abstract (concrete) methods (i.e. method has body)
- Can define both static and instance variables
- Abstract class can also implement interface

When to use which?

- Abstract class is typically used for things that are not "real", but merely a category.
  E.g. Animal is not "real", but Tiger is real. You cannot create an Animal per se, but you can create a Tiger.
  Other possible examples: Fruit, Shape, Instrument, Vehicle, Color
- Interface is good if you know what you must do, but no idea how to do (leave it to teach implemented class to declare body)
  E.g. Printable (must be able to print), Comparable (must be able to compare itself with another object), Database (must store items)
- Abstract is when you partially know what and how to do (i.e. some methods are abstract, but some are given a body).
- Abstract is when there is a clear inheritance relationship for concrete objects, because abstract is meant to be inherited. E.g. Tiger is a Animal.
- Abstract is good if you want to declare non-public members. In interface, all methods are implicitly public
- Abstract class has a strong relationship between super and subclass. E.g. abstract class Shape. class Circle extends Shape. Interface relationship is not as strong. E.g. interface Aircon. class House implements Aircon
- Sometimes something can be considered either abstract or interface. It is up to your judgement
- No matter abstract or interface, the point is for you to do things like Animal animal = new Tiger(); or List list = new ArrayList(); and make use of polymorphism relationship

QUESTIONS?