HW Check during HW quiz ✓9

AP CS Unit 6: Inheritance Notes

Inheritance is an important feature of object-oriented languages. It allows the designer to create a new class based on another class. The new class "inherits" everything the original class has plus it may add additional instance variables and methods.

Let's look at an example.

```
public class Runner {
        public static void main(String[] args) {
                Surprise x = new Surprise();
                if ( x.equals( "ok " ) )
                        System.out.println( "equal" );
                else
                        System.out.println( "not" );

                String s = x.toString();
                System.out.println( s );
        }
}
```

```
public class Surprise {
}
```

This compiles, runs and prints:

```
not
Surprise@19821f (or something similar)
```

But how can it compile and run when the Surprise class has not defined an equals or toString method? Has someone made a terrible mistake? haha ...

No.

The Object class is the root class of all java classes. It has 11 methods that all classes inherit. You need to know 2 of these 11 methods:

public String toString()

public boolean equals(Object o)

Design Issue. How can you tell if one class should be a subclass of another class?

If it has a great deal in common with another class in terms of data and behavior. We can re-use code which is an important concept to cs. add to the super class more specifidity.

Ex 1. If you have an Elephant class and a Mammal class, should one or the other be a subclass of the other? Elephant "is-a" Mammal. thus Elephant is a subclass (or child) of Mammal (super class)

Ex 2. If you have a Room class and a Desk class, should one or the other be a subclass of the other? no, because a Room "has-a" Desk. They are not similar, as a matter of fact desks are data in a room.

General Example of Inheritance.

```
public class Mammal{
        public void speak(){
                System.out.println( "hey" );
        }

        public String toString(){
                return "mammal";
        }
}
```

```
public class Dog extends Mammal{
        public void speak(){
                System.out.println( "woof" );
        }
}
```

Note. The println method is overloaded. When you pass it an object variable, it will call the object's toString method.

What is displayed?

mammal
mammal
hey
wood
false

```
public class Runner {
        public static void main(String[] args) {
                Mammal a = new Mammal();
                Dog d = new Dog();
                System.out.println( a );
                System.out.println( d );
                a.speak();
                d.speak();
                System.out.println( a.equals( d ) );
        }
}
```

The original class is called the Super or parent class. // both used in AP tests

The new class is called the sub-class or child class

The variables and methods of the superclass become part of the subclass; they are inherited by the subclass. The subclass may

- define new istance variables
- define new methods
- redefine inherited instance methods (this is called overriding a method). The headers should be identical (there are some exceptions which we will ignore).

Notice that the keyword extends is used to indicate that a class is a subclass of another. A class can only have one immediate superclass but a given class may have many subclasses.

Go to the java api and examine classes such as the String class and the JButton class.

exer 1-7

This is always tested.

vital & tested ☆

Inheritance and Constructors. The first thing a subclass constructor does is call (either implicitly or explicitly) the constructor for the superclass.

public class AA{
private int a1;
 	
public AA() { match parameters
n = 0;
}
 	
public AA(int c) {
n = c;
}
}
	public class BB extends AA{
	private String str;
	explicit super call
	public BB(String s) {
	super( );
	str = s;
	OR
	// super inserted
	str = s;
	}
	implicit super call
	}
	NOTE only works w/ no-args constructor in parent class!!
		public class CC extends BB{
		private int k;
		
		public CC(int a) {
		super ("word");
		k = a;
		}
		}

Important. If something is private in the superclass, it is not assessible in any subclass.

example: your parents provide you money but you have to ask for some. Methods to ask for data!

Overriding Methods. A subclass may *override* a method from a superclass. If it does, it can still use the superclass's method by use the keyword *super*.

// client code
Pet p = new Pet();
System.out.println(p);
3 years old
 	
Dog d = new Dog("dan");
System.out.println(d);
dan is 3 years old
	public class Pet {
	private int age;
	
	public Pet() {
	age = 3;
	}
	
	public String toString(){
	return age +
	" years old";
	}
	}
		public class Dog extends Pet {
		private String name;
		
		public Dog(String n) {
		super(); // optional
		name = n;
		}
		
		public String toString(){
		String z = super.toString();
		return name + " is " + z;
		}
		}

Why bother with inheritance?

a CS principle.

- Inheritance supports and encourages code reuse. Programmers don't always write entirely new classes; frequently they build on existing classes.
- The built-in Java classes make extensive use of inheritance.

if you use @override, compiler will let you know if the header does not match exactly.

NOTE: c1 == c2 checks if values of ptrs are equal (ie same storage location in memory)

The equals Method, Casting and the instanceof Operator

The equals method in the Object class has the following header:

    public boolean equals(Object obj) ← to override must match exactly

```
// client code

Card c1 = new Card( 3, 7 );
Card c2 = new Card( 3, 7 );
Card c3 = new Card( 2, 11 );
String s = "ok";

System.out.println( c1.equals( c2 ) );
System.out.println( c1.equals( c3 ) );
System.out.println( c1.equals( s ) );

// the above runs and prints:
      true
      false
      false // s not instance of Card
```

```
public class Card {
    private int suit;  // suit = 1, 2, 3, or 4
    private int value; // value = 1 to 13

    public Card( int s, int v ) {
        suit = s;
        value = v;
    }

    public boolean equals( Object x ){
        if ( x instanceof Card == false )
            return false;

        Card c = (Card) x;
        if ( value == c.value && suit == c.suit )
            return true;
        else
            return false;
    }
}
```

programmer/client job

1. The person designing a class decides what it means for two objects to be equal or not. It may be that all their instance variables must be equal or just some.

2. The instanceof operator can be used to test if a variable contains a reference to an object of a specified type.

3. In the statement:

    Card c = (Card) x;

We are assuring the compiler that it is ok to treat the contents of x as a reference to a Card object ~~(and not just a reference to an Object). We are not actually changing the contents of x.~~

To repeat:

1. To override the equals method in the Object class, the new equals methods must have the exact same header as the method in the superclass.

2. Therefore the parameter must be of type Object.

3. However, within the equals method we will want to compare instance variables. The compiler will not let us call the instance variables of the Card class if the parameter is of type Object.

4. Therefore we must cast the variable *x* to the Card class before calling the instance variables.

Note: if you mispell equals or use a parameter of a type other than Object, you won't be overriding equals. If there is no equals, you call Object class equals, which will check ref ptr values for equality.

In general, if an object variable is of type X then you may assign it a reference to an object of type X or a reference to an object that is a subclass of X. For example:

```
public class Fish{
        public void m1(){
                System.out.println("A");
        }
        public void m2(){
                System.out.println("B");
        }
}
```

```
public class Tuna extends Fish{
        public void m1(){
                System.out.println("C");
        }
        public void m3(){
                System.out.println("D");
        }
}
```

Code	Compile Time	Run Time
Fish f = new Tuna();	f is a Fish type & tuna "is a" Fish. OK	no arg Tuna constructor run. OK
f.m1();	f is a Fish type m1 exists in Fish	f will call Tuna classes m1() method. "late-binding"
f.m2();	f is a Fish type m2 exists in Fish	f checks "late-binding" if m2 exists in Tuna, it doesn't so uses M1 classes m2 method
f.m3();	f is a fish type and m3 does not exist in Fish so compile error.	

Tuna x = (Tuna) f;
x.m3();
OR
((Tuna) f).m3();
	Fixes above error.
	x is Tuna type. we cast for
	the compiler to a Tuna object
	& then stuff works for
	compile time!	

↳ type of var determined @ run time.

Tuna t = new Fish( ); does not compile bc a Fish "is-not" a tuna.

Tuna t = new Tuna( ); Compiles bc a Tuna "is-a" Tuna

Fish f = new Fish( ); compiles fine

☆ parent class must be on left! ☆

Note. Calling a method has higher precedence than casting. Consider the following two code snippets

you must wrap the object being cast so it happens first.

BB b = (BB) a.method();	BB b = ((BB) a).method();

On the left, a.method() happens first due to precedence & then casting occurs. method() must exist in a to compile.

On the right, a is cast to BB type and then method() is called so method() may exist only in BB.

Why do something like: Fish f = new Tuna();

1) You may need to override a method from a superclass such as the equals method. In this case you will often pass an argument that contains a reference to a subclass.

```
Student g = new Student( 13 );
Student h = new Student( 13 );
boolean b = g.equals( h );
```

In the last line the argument to the equals method is h, of type Student. We are assigning it to a parameter of type Object, which is the superclass of Student.

Note. If you try to cast something to the wrong type, you may get a compiler error or you may get a runtime error. But you won't get away with it. type protection is strong in java

```
public class Student{
    private int id;

    public Student( int i ){
        id = i;
    }

    public boolean equals( Object x ){
        if ( x instanceof Student ) {
            if ( id == ((Student) x).id )
                return true;
        }
        return false;
    }
}
```

2) You may need an array of objects from the superclass and the subclass.

```
public class Coin{
    private int value;

    public Coin( int v ){
        value = v;
    }

    public int getValue(){
        return value;
    }
}
```

```
public class MagicCoin extends Coin {
    private boolean lucky;

    public MagicCoin( int v ){
        super( v );
        lucky = Math.random() < 0.5;
        System.out.println( lucky );
    }

    public boolean lucky(){
        return lucky;
    }
}
```

In this example, the array elements are of type Coin but may actually contain references to MagicCoin objects as well.

```
public class Runner {
    public static void main(String[] args) {
        Coin [] c = new Coin[10];
        for ( int n=0; n<10; n++ )
            c[n] = get();

        // other code
    }
    private static Coin get(){
        int v = (int)(10*Math.random())+1;
        if ( Math.random() < 0.5 )
            return new Coin( v );
        else
            return new MagicCoin( v );
    }
}
```

coin[] c is holding Coins or MagicCoins!

Here's another example because this topic tends to confuse people.

```
public class Mammal {
    public Mammal() {
        System.out.println("M");
    }

    public void speak(){
        String s = toString();
        System.out.println("I'm a " + s);
    }

    public String toString(){
        return "mammal";
    }
}
```

```
public class Dolphin extends Mammal {
    public Dolphin() {          implicit super()
        System.out.println("D");
    }

    public void swim(){
        String s = toString();
        System.out.println(s + " swimming");
    }

    public String toString(){
        return "dolphin";
    }
}
```

The above classes are fine. The code below compiles and runs except for 2 of the 4 last statements. The blank lines are what is printed out by each statement.

```
public class Runner {
    public static void main(String[] args) {
        Mammal m = new Mammal();                 M
        Dolphin d = new Dolphin();               M      // implicit super() call!
                                                 D
        Mammal md = new Dolphin();               M
                                                 D

        System.out.println( m );                 mamal
        System.out.println( d );                 dolphin
        System.out.println( md );                dolphin   // example of late binding
        doThis( m );                             I'm a mammal
        doThis( d );                             I'm a dolphin
        doThis( md );                            I'm a dolphin
        m.swim();                         C.E.   m is of type mammal - no swim method
        d.swim();                                dolphin swimming
        md.swim();                        C.E.   at compile time MD is a mammal - no swim
        ( (Dolphin) md).swim();                  fooled compiler via casting so works
                                                 dolphin swimming
    }

    public static void doThis(Mammal x){
        x.speak();
    }
}
```

2 of these 4 lines cause compiler errors.

Note: d is-A' Mammal so this compiles

toString from dolphin class is called bc runtime knows via late binding that our object is a Dolphin!

What does it all mean?

(1) If a variable is of type Mammal, you can store a reference to any object that "is a" Mammal. For example:

Mammal md = new Dolphin(); // ok

Dolphin flipper = new Mammal(); // NOT OK

(2) If a method is expecting a Mammal object, the argument can be any object that "is a" Mammal. For example, the doThis method expects a Mammal object which includes any objects of the subclasses of the Mammal class.

public static void doThis(Mammal x){

(3) If a variable is of type Mammal, then you can only call methods of the Mammal class

Mammal md = new Dolphin();
md.speak(); // ok
md.swin(); // NOT OK won't pass compile time
swim

* You may cast the variable to a Dolphin object if you need to call a Dolphin method that is not part of the Mammal class.

((Dolphin)md).swim();

late binding

(4) If a method is overridden, you run the method of the actual object's class (not the class of the variable).

Planet e = new Earth(); // ok because Earth "is a" Planet (I made it a subclass of Planet)
e.m(); // For this to compile, the Planet class must have a *m* method
// If the Earth class overrides the *m* method, then we run the Earth's *m*
* // Otherwise we run the *m* method from the Planet class.

example MC question

Abstract Classes. A big benefit of inheritance is that it allows you to consolidate common code into one class and then extend that class to handle more specific situations.

However, there are some situations where

1. you want to ensure that no objects of that super class are instantiated and/or
2. you do want require every subclass to override a particular method or methods.

In this situation you will create an abstract class by using the keyword abstract.
in front of class

For example. Imagine you are writing a game where different objects exist in a grid. All objects have x and y coordinates and all objects move but different objects move in different ways. You could write a class like this:

has a constructor
all classes that extend must flesh out move
Some methods are fleshed out
	public abstract class Piece{
	private int x, y;
	
	public Piece(int x, int y){
	this.x = x;
	this.y = y;
	}
	
	public abstract void move(int dx, int dy);
	
	public int getX(){
	return x;
	}
	
	// other non-abstract getters and setter
	}

	public class Queen extends Piece{
	public Queen(int x, int y){
	super(x, y);
	}
	
	public void move(int dx, int dy){
	// code
	}
	}

NOPE! cannot construct abstract class	Piece p = new Piece(7, 8);

Can!	Queen q = new Queen(7, 8);
	q.move(5, 6);

Yes, ref type can be abstract but only a 'concrete' class can be constructed. { new Queen }	Piece p = new Queen(7, 8);
	p.move(5, 6);

this works bc Piece also has a move method!