Java Lec 5_Inheritance

Friday, 17 September 2021 6:36 PM

Java_Lec_I...

Basic Java prog.

→ Shapes relation 1-to 3 yr

click 5 buttons → 5 shapes Rect, circle, pallet, Δ

int [1, 2, 3, 4] → Rect circle . - - .

Hierarchy

shape1 shape2 - - .

→ shape[0], - - - .

Industry

Design based

Programming using Java

Java: Inheritance

4 – 10

↳ size of shape

↳ Type of shapes

11 – 16

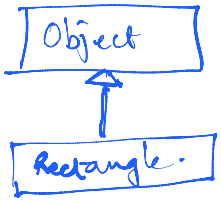
- Mathematical properties (length of side → square, Rectangle)

⋮

17 – 25

- shading

- CG . - - .



Objectives:

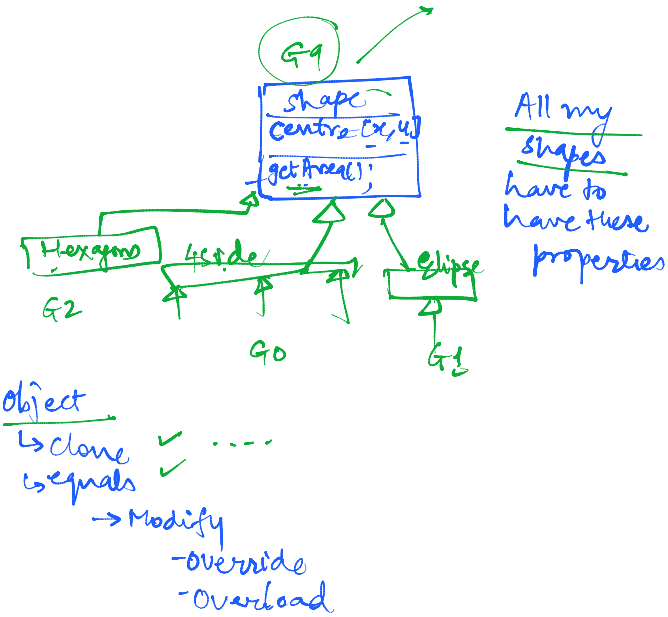
- Get an introduction to Inheritance
- Get a general idea of how a hierarchy of classes is put together

Inheritance: Definition

- **inheritance**: a parent-child relationship between classes

Attributes & methods

- allows sharing of the behavior of the parent class into its child classes
  - one of the major benefits of object-oriented programming (OOP) is this code sharing between classes through inheritance
- child class can add new behavior or override existing behavior from parent



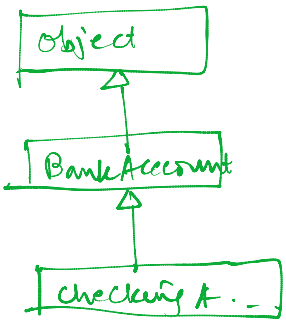
Inheritance terms

- **superclass**, **base class**, **parent class**: terms to describe the parent in the relationship, which shares its functionality
- **subclass**, **derived class**, **child class**: terms to describe the child in the relationship, which accepts functionality from its parent
- **extend**, **inherit**, **derive**: become a subclass of another class

Inheritance in Java

- in Java, you specify another class as your parent by using the keyword `extends`
  - `public class CheckingAccount`
    `extends BankAccount {`

- Java forces a class to have exactly one parent ("single inheritance")
  - other languages (C++) allow multiple inheritance



Bike racing – Assignment 3

Thursday

↳ Hierarchy ? you/rs →

↳ Relations

What is the relationship – 1 way
2 way.

Inheritance in Java

- the objects of your class will now receive all of the state (fields) and behavior (methods) of the parent class
- constructors and static methods/fields are not inherited
- by default, a class's parent is `Object`

Group project

→ TA

23rd Sept

Inheritance Example

```
class BankAccount {
  private double myBal;
  public BankAccount() { myBal = 0; }
  public double getBalance() { return myBal; }
}

class CheckingAccount extends BankAccount {
  private double myInterest;
  public CheckingAccount(double interest) {  }
  public double getInterest() { return
   myInterest; }
  public void applyInterest() {  }
}
```

- `CheckingAccount` objects have myBal and myInterest fields, and getBalance(), getInterest(), and applyInterest() methods

Balance
getBalance()
default
Interest
return
adds to myBal

RecurringDep Acc
Time period -
Monthly pay -

Multiple layers of inheritance

- it is possible to extend a class that itself is a child class; inheritance chains like this can be arbitrarily deep

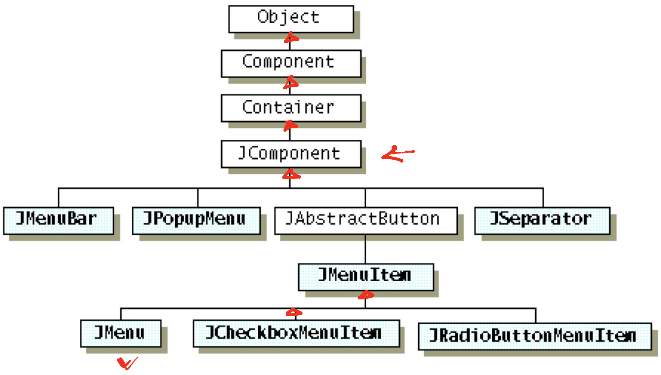
```
public class
  TransactionFeeCheckingAccount
    extends CheckingAccount {
  private static final double FEE =
  2.00;

  public void chargeFee() {
    withdraw(FEE);
  }
}
```


attribute
withdawal fee
→ method

Inheritance Hierarchies

- Deeper layered chain of classes, many children extending many layers of parents



J

"Has-a" Relationships

- "Has-a" relationship: when one object contains another as a field

```
public class BankAccountManager {
  private List myAccounts;
  // ...
}
```

- a `BankAccountManager` object "has-a" `List` inside it, and therefore can use it

BA

Parent
child

CA is a BA.

has a BA

BAM.
List myAccounts;
main()
{ funds int
}

"Is-a" relationships

- "Is-a" relationships represent sets of abilities; implemented through interfaces and inheritance

```
public class CheckingAccount
   extends BankAccount {
   // ...
}
```

- a `CheckingAccount` object "is-a" `BankAccount`
  - therefore, it can do anything an `BankAccount` can do
  - it can be substituted wherever a `BankAccount` is needed
  - a variable of type `BankAccount` may refer to a `CheckingAccount` object

customer of a Bank

BA

is-a → CA balances

→ SA "

→ RD "

→ MF "

Total Assets.

type of array = BankAcc

BA[0] BA[1]

→ for i=0 ...

{ mbalance+=

}

BankAccount ← CheckA - - - -

BankAccountManager

Balance -

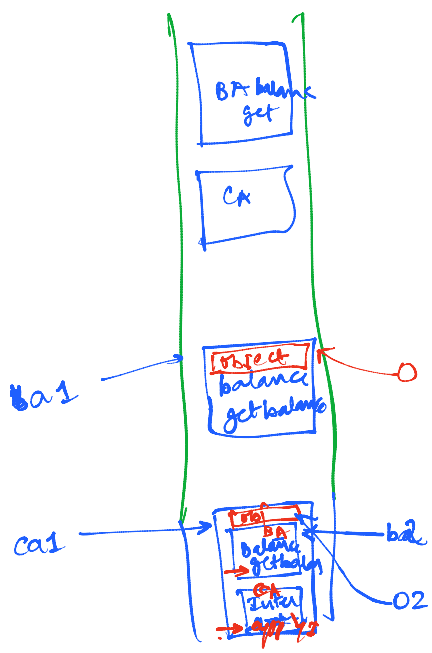
getBalance -

Using the account classes

- `CheckingAccount` inherits `BankAccount`'s methods

```
CheckingAccount c = new
CheckingAccount(0.10);
System.out.println(c.getBalance());
c.applyInterest();
```

-);



all my Accounts

Array of Objects

balance

BA

O1 O2 O3 O4

ba1 ba2 ba3 ba4

O1 = ba1

constructor

→ for i=0; ...

{

}

ba3 = ca1;

ba4 = rd1;

type of object

type of pointer

Using the account classes

- a `BankAccount` variable can refer to a `CheckingAccount` object

pointer of BA

object of CA type

```
BankAccount b2 = new CheckingAccount(0.06);
System.out.println(b2.getBalance());
```

- an `Object` variable can point to either account

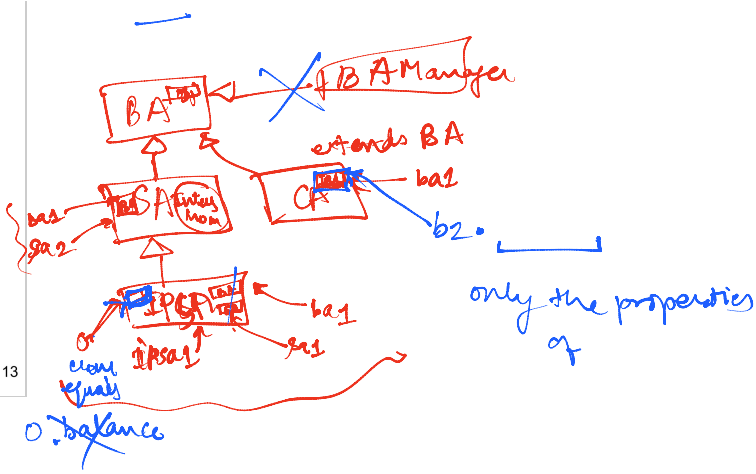
BAManager

```
BankAccount b2 = new CheckingAccount(0.06);
System.out.println(b2.getBalance());
```

- an `Object` variable can point to either account type

```
Object o  = new BankAccount();
Object o2 = new CheckingAccount(0.09);
```

class Object ← anything



Some code that won't compile

- `CheckingAccount` variable can't refer to `BankAccount` (not every `BankAccount` "is-a" `CheckingAccount`)

```
CheckingAccount c = new BankAccount();
```

- cannot call a `CheckingAccount` method on a variable of type `BankAccount` (can only use `BankAccount` behavior)

```
BankAccount b = new CheckingAccount(0.10);
b.applyInterest();
```

- cannot use any account behavior on an `Object` variable

```
Object o = new CheckingAccount(0.06);
System.out.println(o.getBalance());
o.applyInterest();
```

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overriding behavior

- Child class can replace the behavior of its parent's methods by redefining them

```
public class BankAccount {
  private double myBalance;  // ....
  public String toString() {
    return getID() + " $" + getBalance();
} }

public class FeeAccount extends BankAccount {

  private static final double FEE = 2.00;
  public String toString() {  // overriding
    return getID() + " $" + getBalance()
      + " (Fee: $" + FEE + ")";
} }
```

Overriding behavior example

```
BankAccount b = new BankAccount("Ed",9.0);
FeeAccount f = new FeeAccount("Jen", 9.0);

System.out.println(b);
System.out.println(f);
```

- Output:

```
Ed $9.0
Jen $9.0 (Fee: $2.0)
```


Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Access modifiers

- `public`: visible to all other classes
  ```
  public class BankAccount
  ```
- `private`: visible only to the current class, its methods, and every instance (object) of its class
  - a child class cannot refer to its parent's private members!
  ```
  private String myID;
  ```
- `protected` (this one's new to us): visible to the current class, and all of its child classes
  ```
  protected int myWidth;
  ```
- package (default access; no modifier): visible to all classes in the current "package" (seen later)
  ```
  int myHeight;
  ```


Access modifier problem

```
public class Parent {
  private int field1;
  protected int field2;
  public int field3;
  private void method1() {}
  public void method2() {}
  protected void setField1(int
 value) {
    field1 = value;
  }
}
```

Access modifier problem

```
public class Child extends Parent {
  public int field4;

  public Child() {        // Which are legal?
    field4 = 0;           // __________
    field1++;             // __________
    field2++;             // __________
    field3++;             // __________
    method1();            // __________
    method2();            // __________
    setField1(field4);    // __________
  }
}
```


Some code that won't compile

```
public class Point2D {
  private int x, y;
  public Point2D(int x, int y) {
    this.x = x;   this.y = y;
  }
}

public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    this.x = x;  this.y = y;  // can't do
  this!
    this.z = z;
  }
}
```

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

`super` and constructors

- if the superclass has a constructor that requires any arguments (not `()`), you *must* put a constructor in the subclass and have it call the super-constructor (call to super-constructor must be the first statement)

`super` and constructors

```
public class Point2D {
  private int x, y;

  public Point2D(int x, int y) {
    this.x = x;   this.y = y;
  }
}

public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);   // calls Point2Dconstructor
    this.z = z;
  }
}
```


`super` keyword

- used to refer to superclass (parent) of current class
- can be used to refer to parent class's methods, variables, constructors to call them
  - needed when there is a name conflict with current class
- useful when overriding and you want to keep the old behavior but add new behavior to it

- syntax:

```
super(args);              // call parent's constructor
super.fieldName           // access parent's field
super.methodName(args);   //   or method
```

`super` example

```
public class BankAccount {
  private double myBalance;
  public BankAccount() { myBalance = 0; }
  public double getBalance() { return
  myBalance; }
  public void withdraw(double amount) {
    myBalance -= amount;
} }
public class FeeAccount
                    extends BankAccount {
  public void withdraw(double amount) {
    super.withdraw(amount);
    if (getBalance() < 100.00)
      withdraw(2.00); // charge $2 fee
} }
```

- Can the withdraw(2.00) cause any problems?

Constructors in extended classes

- A constructor of the extended class can invoke one of the superclass's constructors by using the *super* method.
- If no superclass constructor is invoked explicitly, then the superclass's no-arg constructor

  `super( )`

  is invoked automatically as the first statement of the extended class's constructor.
- Constructors are not methods and are NOT inherited.

Three phases of an object's construction

- When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a three-phase construction:
  - invoke a superclass's constructor
  - initialize the fields by using their initializers and initialization blocks
  - execute the body of the constructor
- The invoked superclass's constructor is executed using the same three-phase constructor. This process is executed recursively until the `Object` class is reached

To Illustrate the Construction Order. . .

```
class X {
     protected int xOri = 1;
     protected int whichOri;
     public X() {
          whichOri = xOri;
     }
}
```

```
class Y extends X {
     protected int yOri = 2;
     public Y() {
          whichOri = yOri;
     }
}
```

```
Y objectY = new Y();
```

Step	what happens	xOri	yOri	whichOri
0	fields set to default values	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	Object constructor invoked	0	0	0
4	X field initialization	1	0	0
5	X constructor executed	1	0	1
6	Y field initialization	1	2	1
7	Y constructor executed	1	2	2

Overloading and Overriding Methods

- *Overloading*: providing more than one method with the same name but different parameter list
  - overloading an inherited method means simply adding new method with the same name and different signature
- O*verriding*: replacing the superclass's implementation of a method with your own design.
  - both the parameter lists and the return types must be exactly the same
  - if an overriding method is invoked on an object of the subclass, then it's the subclass's version of this method that gets implemented
  - an overriding method can have different access specifier from its superclass's version, but only wider accessibility is allowed
  - the overriding method's `throws` clause can have fewer types listed than the method in the superclass, or more specific types

Fields/Methods in Extended Classes

- An object of an extended class contains two sets of variables and methods
  1. fields/methods which are defined locally in the extended class
  2. fields/methods which are inherited from the superclass

?

Accessibility and Overriding

- a method can be overridden only if it's accessible in the subclass
  - `private` methods in the superclass
    - cannot be overridden
    - if a subclass contains a method which has the same signature as one in its superclass, these methods are totally unrelated
  - `package` methods in the superclass
    - can be overridden if the subclass is in the same package as the superclass
  - `protected`, `public` methods
    - always will be

Not as that simple as it seems!

```
package P1;

public class Base {
   private void pri( ) { System.out.println("Base.pri()"); }
           void pac( ) { System.out.println("Base.pac()"); }
   protected void pro( ) { System.out.println("Base.pro()"); }
   public void pub( ) { System.out.println("Base.pub()"); }

   public final void show( ) {
       pri();  pac();  pro();  pub(); }
}

package P2;

import P1.Base;

public class Concrete1 extends Base {
   public void pri( ) { System.out.println("Concrete1.pri()"); }
   public void pac( ) { System.out.println("Concrete1.pac()"); }
   public void pro( ) { System.out.println("Concrete1.pro()"); }
   public void pub( ) { System.out.println("Concrete1.pub()"); }
}
```

```
Concrete1 c1 = new Concrete1();
c1.show( );
```

Output?

```
Base.pri()
Base.pac()
Concrete1.pro()
Concrete1.pub()
```

Sample classes (cont.)

```
package P1;

import P2.Concrete1;

public class Concrete2 extends Concrete1 {
   public void pri( ) { System.out.println("Concrete2.pri()"); }
   public void pac( ) { System.out.println("Concrete2.pac()"); }
   public void pro( ) { System.out.println("Concrete2.pro()"); }
   public void pub( ) { System.out.println("Concrete2.pub()"); }
}
```

```
Concrete2 c2 = new Concrete2();
c2.show( );
```

Output?

```
Base.pri()
Concrete2.pac()
Concrete2.pro()
Concrete2.pub()
```

Sample classes (cont.)

```
package P3;
import P1.Concrete2;
public class Concrete3 extends Concrete2 {
   public void pri( ) { System.out.println("Concrete3.pri()"); }
   public void pac( ) { System.out.println("Concrete3.pac()"); }
   public void pro( ) { System.out.println("Concrete3.pro()"); }
   public void pub( ) { System.out.println("Concrete3.pub()"); }
}
```

```
Concrete3 c3 = new Concrete3();
c3.show( );
```

Output?

```
Base.pri()
Concrete3.pac()
Concrete3.pro()
Concrete3.pub()
```

Hiding fields

- Fields cannot be overridden, they can only be hidden
- If a field is declared in the subclass and it has the same name as one in the superclass, then the field belongs to the superclass cannot be accessed directly by its name any more

Polymorphism

- An object of a given class can have multiple forms: either as its declared class type, or as any subclass of it
- an object of an extended class can be used wherever the original class is used
- **Question**: given the fact that an object's actual class type may be different from its declared type, then when a method accesses an object's member which gets redefined in a subclass, then which member the method refers to (subclass's or superclass's)?
  - when you invoke a method through an object reference, the *actual class of the object* decides which implementation is used
  - when you access a field, the declared type of the reference decides which implementation is used

Output?

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

```
class SuperShow {
   public String str = "SuperStr";

   public void show( ) {
       System.out.println("Super.show:" + str);
   }
}

class ExtendShow extends SuperShow {
   public String str = "ExtendedStr";

   public void show( ) {
       System.out.println("Extend.show:" + str);
   }

   public static void main (String[] args) {
      ExtendShow ext = new ExtendShow( );
      SuperShow sup = ext;
      sup.show( );   //1
      ext.show( );   //2
      System.out.println("sup.str = " + sup.str);      //3
      System.out.println("ext.str = " + ext.str);//4
   }
}
```

methods invoked through object reference

field access

`protected` members

- To allow subclass methods to access a superclass field, define it `protected`. But be cautious!
- Making methods `protected` makes more sense, if the subclasses can be trusted to use the method correctly, but other classes cannot.

What `protected` really means

Precisely, a `protected` member is accessible

- Within the class itself
- within code in the same package
- it can also be accessed from a class through object references that are of at **least the same type as the class** – that is , references of the class's type or one of its subtypes

What `protected` really means

```
public class Employee {
     protected Date hireDay;
     . . .
}

public class Manager extends Employee {
     . . .
     public void printHireDay (Manager p) {
        System.out.println("mHireDay: " +
                           (p.hireDay).toString());
     }
     // ok! The class is Manager, and the object reference type is also Manager



     public void printHireDay (Employee p) {
       System.out.println("eHireDay: " +
                     (p.hireDay).toString());
     }
     // wrong! The class is Manager, but the object reference type is Employee
     // which is a supertype of Manager
     . . .
}
```

`super` example

```
public class BankAccount {
  private double myBalance;
  public BankAccount() { myBalance = 0; }
  public double getBalance() { return myBalance;
  }
  public void withdraw(double amount) {
    myBalance -= amount;}
   public String toString() {
    return getID() + " $" + getBalance();
} }

public class FeeAccount
                    extends BankAccount {
  public void withdraw(double amount) {
    super.withdraw(amount);
    if (getBalance() < 100.00)
      super.withdraw(2.00); // charge $2 fee
} }
```


Which method gets called?

```
BankAccount b = new FeeAccount("Ed", 9.00);
b.withdraw(5.00);
System.out.println(b.getBalance());
```

- Will it call the `withdraw` method in `BankAccount`, leaving Ed with $4?
- Will it call the `withdraw` method in `FeeAccount`, leaving Ed with $2 (after his $2 fee)?

The answer: dynamic binding

- The version of `withdraw` from `FeeAccount` will be called
- The version of an object's method that gets executed is always determined by that object's type, *not* by the type of the variable
- The variable should only be looked at to determine whether the code would compile; after that, all behavior is determined by the object itself

Static and Dynamic Binding

- **static binding**: methods and types that are hard-wired at compile time
  - static methods
  - referring to instance variables
  - the types of the reference variables you declare
- **dynamic binding**: methods and types that are determined and checked as the program is running
  - non-static (a.k.a virtual) methods that are called
  - types of objects that your variables refer to

Polymorphism

- inheritance provides a way to achieve polymorphism in Java
- **polymorphism**: the ability to use identical syntax on different data types, causing possibly different underlying behavior to execute
  - example: If we have a variable of type `BankAccount` and call `withdraw` on it, it might execute the version that charges a fee, or the version from the checking account that tallies interest, or the regular version, depending on the type of the actual object.

Type-casting and objects

- You cannot call a method on a reference unless the reference's type has that method

```
Object o = new BankAccount("Ed",9.00);
o.withdraw(5.00);  // doesn't compile
```

- You can cast a reference to any subtype of its current type, and this will compile successfully

```
((BankAccount)o).withdraw(5.00);
```

Converting Between Subclass and Superclass Types

- Occasionally you need to convert from a superclass reference to a subclass reference

```
BankAccount anAccount = (BankAccount) anObject;
```

- This cast is dangerous: if you are wrong, an exception is thrown

The `instanceof` keyword

- Performs run-time type check on the object referred to by a reference variable
- Usage: *object-reference* `instanceof` *type* (result is a boolean expression)
  - if *type* is a class, evaluates to **true** if the variable refers to an object of *type* or any subclass of it.
  - if *type* is an interface, evaluates to true if the variable refers to an object that implements that interface.
  - if ***object-reference* is null, the result is false**.
- Example:

```
Object o = myList.get(2);
if (o instanceof BankAccount)
  ((BankAccount)o).deposit(10.0);
```

Converting Between Subclass and Superclass Types

- Solution: use the `instanceof` operator
- `instanceof`: tests whether an object belongs to a particular type

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Down-casting and runtime

- It is illegal to cast a reference variable into an unrelated type (example: casting a `String` variable into a `BankAccount`)
- It is legal to cast a reference to the wrong subtype; this will compile but crash when the program runs
  - Will crash even if the type you cast it to has the method in question

```
((String)o).toUpperCase();        // crashes
((FeeAccount)o).withdraw(5.00);   // crashes
```

Some `instanceof` problems

```
Object o = new BankAccount(...);
BankAccount c = new CheckingAccount(...);
BankAccount n = new NumberedAccount(...);
CheckingAccount c2 = null;
                                        T/F
  ???
o instanceof Object
  ________
o instanceof BankAccount
  ________
o instanceof CheckingAccount
  ________
c instanceof BankAccount
  ________
c instanceof CheckingAccount
```


A dynamic binding problem

```
class A {
  public void method1() { System.out.println("A1"); }
  public void method3() { System.out.println("A3"); }
}

class B extends A {
  public void method2() { System.out.println("B2"); }
  public void method3() { System.out.println("B3"); }
}

A var1 = new B();
Object var2 = new A();

var1.method1();_____________
var1.method2();____________
var2.method1();_____________
((A)var2).method3();_________________
```

OUTPUT???

The Object & Class Classes

Object Class : Top class in Java

equals() method

boolean equals(Object obj)

getClass() method

Class getClass()

toString() method

String toString()

Class Class

getName() method

String getName()

getSuperClass() method

Class getSuperClass()

forName() method

Static Class forName (String cIsName) throws ClassNotFoundException

The Object & Class Classes

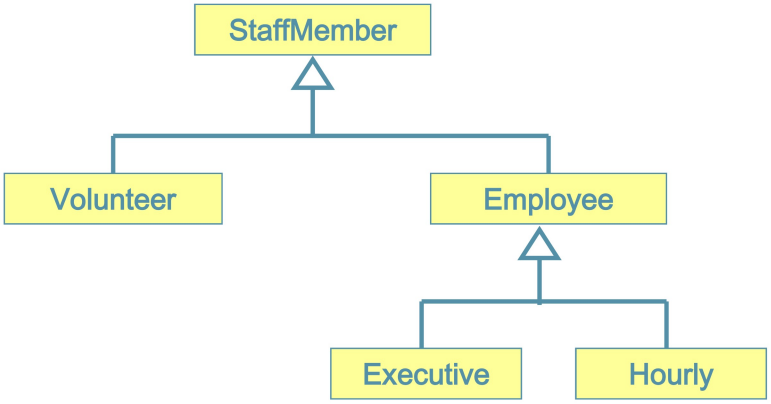
```
class ClassDemo {

  public static void main(String args[]) {

    Integer obj = new Integer(8);
    Class cls = obj.getClass();
    System.out.println(cls);
  }
}
```

Result :

Class java.lang.Integer

A payment problem

- **Consider the following class hierarchy: How to pay all staff members using their pay() methods.**