Inheritance

Object Relationships
super - superclass Object
Extending a class
Methods in subclass
Using this reference
Accessing superclass Instance variables
Using protected access
Subclass construction
Multiple constructors
Method overloading

Account → SAccount

In your last assignment you added new functionality and instance variables to Account class to create a more specialised SAccount class.
- withdraw() method was changed
- a new method addInterest(double rate) was added.
- a new instance variable minAmount was added
- constructor was changed to take an additional argument
- no change in methods such as deposit() and getBalance()
Could have used inheritance!

```java
class SAccount extends Account {
    new and redefined methods
    new instance variables;
}
```

- Add a new method `addInterest(double rate)`
- Redefine `withdraw()`
- add Instance variable `minAmount`
- write the constructor

Whenever I change superclass implementation it will be reflected in the subclasses. No worries !!!

---

The supreme superclass `Object`

Every class (even one that does not use `extends`) is a subclass of `Object`.

Why?

Object class has a small number of methods that are useful for all classes.

- `equals()`
- `toString()`
- `clone()`
Object Oriented Relationships

Two commonly used clauses in class relationship

• is-a ← Inheritance relationship
• has-a ← Composition relationship

A home is a house that has a family and a pet.

If House, Family and Pet are existing classes then in java we write

```java
public class Home extends House {
    Family inhabitants;
    Pet thePet;
}
```

Guess the relationship between the classes

a) Manager and Employee
b) Project and Manager
c) Person and Student
d) Book and Author

How would you write them in Java?
A Re-look at Account class

```java
public class Account {
    public Account(...
    public double getBalance(..
    public String getID(..
    public String getName(..
    public void deposit(...
    public void withdraw(...
    public void print(...
    public void transfer(...
    private String accID;
    private double balance;
    private String name;
}
```

Well Encapsulated

Note transfer uses other operations of the class - need not check again.

```java
public boolean transfer(Account account, double amount) {
    if (withdraw(amount)) {
        account.deposit(balance);
        return true;
    }
    else return false;
}

void print() {
    System.out.println("Account ID = " + accID);
    System.out.println("Name = " + name);
    System.out.println("Balance = "+balance);
}
```
Extending the Account class

class SAccount extends Account
{
    public SAccount(String accountID,String accountName,double amount,double minAmount){
        super(accountID, accountName, amount);
        this.minAmount = minAmount;
    }
    public SAccount(String accountID,
                    String accountName, double amount)   {
        this(accountID,accountName,amount,0.0);
    }
    public double getMinAmount() {
        return minAmount;
    }
}  

public void addInterest(double rate)   {
    deposit(getBalance() * rate/100);
}

void print()   {
    super.print();
    System.out.println("Min. Amount = " + minAmount);
}

public boolean withdraw(double amount) {
    if (getBalance() >= amount + minAmount) {
        super.withdraw(amount);
        return true;
    } else return false;
}

private double minAmount;

Two constructors
Two constructors
Accessor for subclass instance variable
New method
Overridden print()
Overridden withdraw()
Asking for trouble ...

What if I leave out super in super.withdraw(amount) ?

```java
public boolean withdraw(double amount) {
    if (getBalance() >= amount + minAmount) {
        super.withdraw(amount);
        return true;
    }
    else return false;
}
```

Result is a recursive call!
An infinite loop ... cause program to hang

Which method is called ?

SAccount sAcc1 = new SAccount("s12345","Graham",1000,800);

sAcc1.deposit(100); ← Not overridden. Superclass method called

sAcc1.withdraw(500); ← Overridden. Subclass method called

sAcc1.addInterest(0.5); ← New method. Subclass method called
Using the **this** reference

this refers to methods and instance variables of current object.

class SAccount extends Account
{
    public SAccount(String accountID, String accountName, double amount, double minAmount){
        super(accountID,accountName,amount);
        this.minAmount = minAmount;
    }

    private double minAmount;
}

Instance variable of superclasses

- Instance variables are automatically inherited by subclasses.
- But if they are private they cannot be accessed directly.
- Hence only way to change them is through superclass mutators.

Subclass object

Inherited (Superclass) instance variables (private)
(access through superclass mutators)

Subclass instance variables (methods can access them directly)
What’s wrong with SAccount withdraw() below?

class SAccount extends Account
{
    public boolean withdraw(double amount) {
        if (balance >= amount + minAmount)
        {
            balance -= amount;
            return true;
        }
        else return false;
    }

    private double minAmount;
}

balance is a private instance variable of superclass Account !!!

Protected Access

• protected alternative to private and public

• If an instance variable is declared protected it can be accessed by methods of that class, its subclasses and all other classes within the same package (or directory).

• However, it cannot be accessed by other class users.

• Next Sample code illustrates the difference between the different access specifiers.
To use protected or not?

Many programmers use the **protected** feature as it strikes a balance between **absolute protection** and **no protection** at all.

However, they break the **encapsulation rule** as the designer of the **superclass** has no control over the authors of the **subclass**.

Furthermore, classes with protected data are hard to modify as someone may have written a subclass based on it and may have accessed the protected data directly.
Quiz The **Date** class designed by *programmer A* was used by *programmer B* in class **Meeting**.
Now B claims A's Date class has bugs.
Comment.

class Date { // Designed by programmer A
    Date(int d, int m, int y) {
        day = d; month = m; year = y;
    }
    // advances d days taking care of month and year
    void advance(int d) {
        // ... } } void print() {
        System.out.println(""+day+"/"+month+"/"+year); } protected int day;
protected int month;
protected int year;
}  

public class Meeting { // designed by programmer B
    Meeting(String c, String t, Date d, String v) { 
        chairman = c; title = t; date = d; venue = v;
    }
    void postpone(int days) {
        date.day += days;
    }
    void print() {
        System.out.println(title+" "+chairman+" "+venue);
        date.print();
    }

    public static void main(String arg[]) { 
        Date d = new Date(28,8,2001);
        Meeting meet=new Meeting("Tom","AGM",d,"9.8");
        meet.postpone(7);
        meet.print();
    }
private String title;
private String chairman; Composition
private String venue; Relationship
private Date date;
Subclass Construction

whenever a subclass object is constructed, the superclass constructor must be called.

syntax used: keyword super followed by construction parameters if any

must be the first statement in the method

If we omit this statement compiler looks for a subclass constructor with no arguments – default constructor.

```
public SAccount(String accountID, String accountName, double amount, double minAmount)
{
    super(accountID, accountName, amount);
    this.minAmount = minAmount;
}
```

Multiple Constructors

Suppose we have a Fraction class with two instance variables numerator and denominator. Naturally the constructor will take two arguments to set these instance variables.

To create a Fraction 3/4 we can call new Fraction(3,4)

How about creating 3 (same as 3/1) ? New Fraction(3,1)

But the designers of Fraction class may provide another constructor which takes only one argument the value for numerator making it convenient for users - new Fraction(3)

```
public Fraction (int num, int den) {
    numerator = num;
    denominator = den;
}
```

```
public Fraction (int num) {
    numerator = num;
    denominator = 1;
}
```
SAccount Multiple Constructors

Most accounts have minimum amount set to 0. Why not provide a constructor which will set it to 0 if no value is passed for min Amount?

SAccount = new SAccount("s1234","Sam", 120, 100);
SAccount = new SAccount("s1235","Tom", 3500, 0);
SAccount = new SAccount("s1236","Tom", 4200, 0);
SAccount = new SAccount("s1237","Tom", 3800, 0);

We can provide an additional constructor that will use the services of the first one with value for minAmount set to 0.

// note only 3 arguments are passed
public SAccount(String accountID, String accountName, double amount){
   this(accountID,accountName,amount,0.0);
}

Method Overloading

• To attract wealthy customers new CEO of Bank XYZ starts giving new saving account holders stepwise interest rates.
• one rate for the first $2,000 and another rate for amounts in excess of $2,000.
• This would mean we would need another method that takes the two different rates as arguments.
• SAccount oldFellow = new SAccount(........);
• SAccount newFellow = new SAccount(........);
• oldFellow.addInterest(1.1);
• newFellow.addInterest(1.1, 1.2);