Abstract Classes

Suppose we want write a program that manipulates various types of bank accounts.

An Account typically has following features;

Name, AccountNumber, Balance.

Following operations can be performed:

Deposit, Withdraw, toString

However, a Bank does not offer a single type of account.

We can have Savings Account, Checking Account, etc.

They share several common features, and have some differences.
SavingsAccount: There are restrictions on Withdraw. Allows only 3 withdraws in a month. Minimum balance must be maintained.

No such restrictions to withdraw from Checking Account.

Savings account gives interest. Checking account does not give any interest.

How do we design classes?

One single class Account?

Two different classes: CheckingAcc and SavingAcc?
One single class Account.

```java
public class Account
    private String Name;
    private String AccNum;
    private int balance;
    private String AccType;
    private double InterestRate;

    /* Constructors and methods for deposit, withdraw, toString... */

    // Method for Withdraw.
    public void withdraw(int amount) {
        // Check if the type is saving or Checking..
        // Do appropriate operations.
    }

    public void calcInterest() {
        // Do this only if the account is Saving.
    }
```
Satisfactory design?

What if there are actually 10 different types of accounts each with different ways of doing withdraw?

huge switch statement?

What if the bank decides to offer a new type of account?

Go into the code and make changes to it?
How about creating a class for each account? SavingAcc CheckingAcc?

public class SavinAcc

public class CheckingAcc

Design and implement each class completely separately?

However, there are several common features. Code for deposit(), toString() is the same.

Moreover, there is a relation among two accounts.

Each one of them is an Account.

We want to maintain that relation.

They are not two completely unrelated classes.
First design the class: CheckingAcc.

SavingAcc extends CheckingAcc. Can override withDraw() method and add calculateInt() method.

We have accomplished code re-use.

Disadvantages?
Now it means every SavingAcc is also a CheckingAcc!!
SavingAccount and CheckingAccount have some features that are common while some features are different.

Both of them are accounts.

Create an interface named Account.

SavingsAcc and CheckingAcc implements Account?

Account has methods withdraw, deposit, toString etc.
Disadvantages?

Code duplication. Method deposit is same for both classes.

In future, changing implementation of deposit in SavingsAcc may also cause to change implementation of deposit in CheckingAcc.
A better alternative.

Create a Class Named Account.

Extend it with SavingsAcc and CheckingAcc.

How should the class Account look like?
public class Account {

    private String Name;
    private String AccNum;
    private int Balance;

    /* Constructor */

    Code for constructor
    ....
    ....

    /* Method deposit */

    public void deposit(double amount) {
        balance = balance + amount;
    }

    /* Method toString */

    Code for toString
    ....
    ....

    /* Method for withDraw */

    Code for withDraw.
    ...

}
Well, what should be the code for `withDraw()`?

We do not have sufficient information at this point.

The code for this method depends on the type of the account.

How do we handle this?

Similar for `toString`
How about the following?

```java
public void withDraw(int amount) {
}
```

Method with empty body!!!!

In classes SavingAcc and CheckingAcc override the method withDraw().

Rely on Polymorphism.

This way of doing things is almost good..
With this one can write a driver program that has the following statement.

```
Account a = new Account(----, ----, ----);
a.withDraw(100);
```

This is fine. But what does this mean?

We are creating an Account.

However, there is no such concrete thing as Account.

We have Checking Account, Savings Account.

When you go to a bank. You don’t just open and Account.

You open either Checking or Saving Account.

What is the meaning of `a.withDraw()`?
Account is an abstract concept.

We know intuitively what it means. But we do not have complete information.

SavingsAccount and CheckingAccount are more concrete.

Now we have complete information.

This leads to the notion of abstract classes.
Account is an **abstract** concept.

We know intuitively what it means. But we do not have complete information.

*SavingsAccount* and *CheckingAccount* are more concrete.

Now we have complete information.

This leads to the notion of **abstract classes**.
abstract public class Account {

    private String Name;
    private String AccNum;
    private int Balance;

    /* constructor */

    /* Methods that can be defined properly */

    public void deposit(int amount) {
        ....
        ...
        Write code
        ...
    }

    abstract public void withDraw(int amount);

    abstract public void calcInterest();

    abstract public String toString();
}
abstract public class Account

We are defining an abstract class with name Account

abstract is a keyword.

abstract public void withDraw(int amount);

This class has an abstract method named withDraw().

Note the “;” at the end.

This method has no body.

An abstract method should not have any body.

It will be overridden in a class that extends Account.
Account is an abstract class.

We can not create any objects of an abstract class.

The following is not allowed.

Account a = new Account(---, ---. ---);

There is no such concrete thing as Account.

So there are no objects of the class Account.

An abstract generally have some abstract methods and some non-abstract (also called concrete) methods.

It is not necessary that an abstract class must have an abstract method.
public class SavingsAcc extends Account {

    int NumWithdrawals;
    double InterestRate;

    /*Override the withDraw method now */

    public void withdraw(int amount) {
        ...
        ...
        Actual Code for the method
    }

    public void calcInterest() {

        ....
        Code for the method
        ...
    }

    public String toString() {

        ....
        Code for toString
        ...
    }
}
public class CheckingAcc extends Account {

    /*Override the withDraw method now */

    public void withDraw(int amount) {
        ...
        ...
        Actual Code for the method
    }

    public void calcInterest() {
        ....
        Code for the method
        ...
    }

    public String toString() {
        ....
        Code
        ....
    }
}
CheckingAcc and SavingAcc extend Account

They are not abstract classes.

So they must override the all the abstract methods in Account. And these methods must be non-abstract methods.

If A is an abstract class and B is a non abstract that extends A, then B must override all abstract methods in A. These methods must be non-abstract methods.

However, if B itself is an abstract class, then it need not override abstract methods of A.
Now we can create objects of type CheckingAcc and SavingAcc

SavingsAcc s = new SavingsAcc();

CheckingAcc a = new CheckingAcc();

s.withDraw() and a.withDraw() work the way they are supposed to work.
Now there is a sibling relation between CheckingAcc and SavingAcc.

If the bank offers a new type of Account, then we can create a class corresponding to that account.
Account a = new Account() is not allowed.

SavingsAccount s = new SavingsAccount() is allowed.

Account a = new SavingsAccount() is allowed.

Type of a is Account. It is pointing to an object of type SavingsAccount.

Since SavingsAccount extends Account, SavingsAccount is a subtype of Account.

So a variable of type Account can point to an object of type SavingsAccount.

Though there are no objects of type Account.

Useful in writing programs that deal with all accounts (i.e., irrespective of different types of account).
Account a = new SavingsAccount()

Account b = new CheckingAccount().

a.deposit(500); b.deposit(400)

**method** deposit is defined in class Account. Its not overridden in its subclasses.

a.withdraw(500); b.withdraw(200)

**Which withdraw is executed?**

**Polymorphism:** Look at the object type.
Account a;
SavingsAccount s;
CheckingAccount c;

a = new SavingsAccount();
s = new Account();
c = new savingsAccount();

a = s;
s = a;
s = c;

Which of the statements are valid?
More examples:

SalariedEmployee, HourlyEmployee:

Abstract class would be Employee.