Inheritance

Some of the material presented is adapted from Java Software Solutions by John Lewis.

A class is to an object what a blueprint is to a house.
A class establishes the characteristics and behaviours of an object but reserves no memory space for variables (unless those variables are declared as `static`). Classes are the plan, and objects are the embodiment of that plan.

Many houses can be created from the same blueprint. They are essentially the same house in different locations with different people living in them.

Now suppose you want a house that is similar to another but with some different or additional features. You want to start with the same basic blueprint but modify it to suit new, slightly different, needs. Many housing developments are created this way. The houses in the development have the same core layout, but they have unique features. For instance, they might all be split-level homes with the same basic room configuration, but some have a fireplace or full basement while others do not, or an upgraded gourmet kitchen instead of the standard version.

It's likely that the housing developer commissioned a master architect to create a single blueprint to establish the basic design of all houses in the development, then a series of new blueprints that include variations designed to appeal to different buyers. The act of creating the series of blueprints was simplified since they all begin with the same underlying structure, while the variations give them unique characteristics that may be important to the prospective owners.

Creating a new blueprint that is based on an existing blueprint is analogous to the object-oriented concept of ***inheritance*** , which is the process in which a new class is derived from an existing one.

Inheritance is a powerful software development technique and a defining characteristic of object-oriented programming.

Via inheritance, the new class automatically contains (inherits) the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class or modify the inherited ones.

In general, new classes can be created via inheritance faster, easier, and cheaper than by writing them from scratch. Inheritance is one way to support the idea of ***software reuse***. By using existing software components to create new ones, we capitalize on the effort that went into the design, implementation, and testing of the existing software.

Single Inheritance

In single inheritance, subclasses inherit the features of one superclass. Figure 1 gives an illustration of such a relationship.

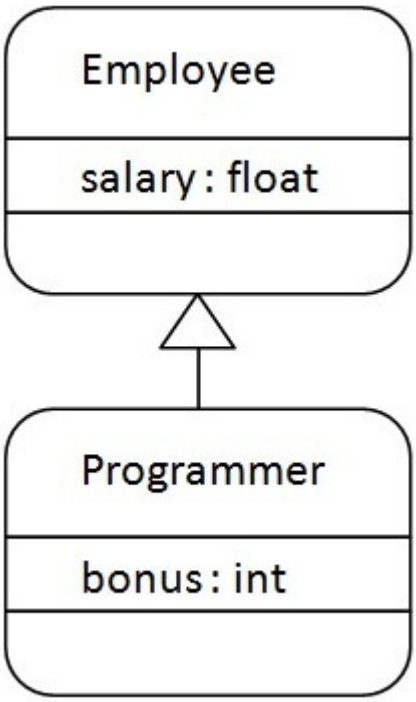
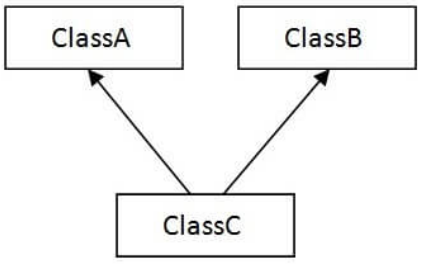


Figure 1: UML diagram illustrating inheritance

NOTE: Some languages, such as C++, support ***multiple inheritance***, which means that the deriving (or subclass) can inherit from multiple derived classes (or superclasses). Luckily, we don't have to worry about multiple inheritance (and the various design problems it can create) in Java.



Multiple Inheritance (not allowed in Java)

In figure 1 the following statements can be made:

- Employee is the **superclass** (sometimes called the **base class** or **parent class**)
- Programmer is the **subclass** (sometimes called the **derived class** or **child class**)
- Inheritance is said to provide an ***IS-A*** relationship. A *Programmer IS-A Employee.*
  Side note: This differs from a concept called **Composition** which is said to have a ***HAS-A*** relationship. For example, A *Person HAS-A DateOfBirth*. More on this later.
-

It is important to note, however, that single inheritance can have multiple levels ('single' means that a class can only inherit a single class (as opposed to multiple inheritance. It does not mean you can only have a single level of inheritance.)

Figure 2 shows examples of the possible inheritance hierarchies

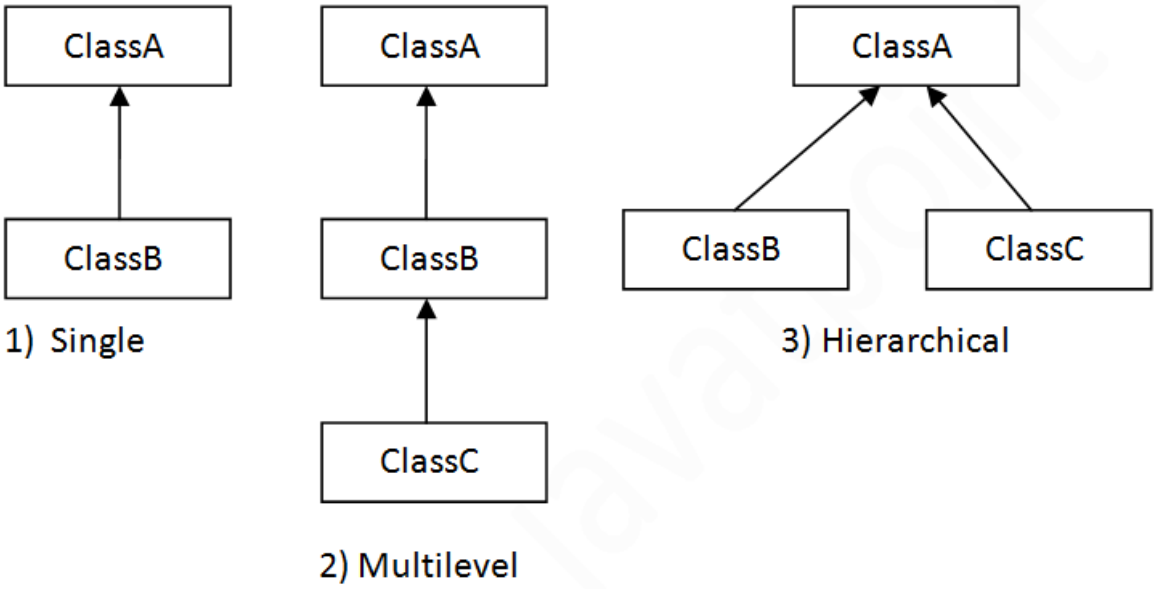


Figure 2: Possible hierarchies via (single) inheritance.

Let's get practical (A BankAccount system)

Before we get into this, I gave an example in the lecture of a **Student – Person inheritance relationship** (A Student IS-A Person). You can find related code in **Appendix A**.

We will look at an inheritance hierarchy concerning BankAccount objects.

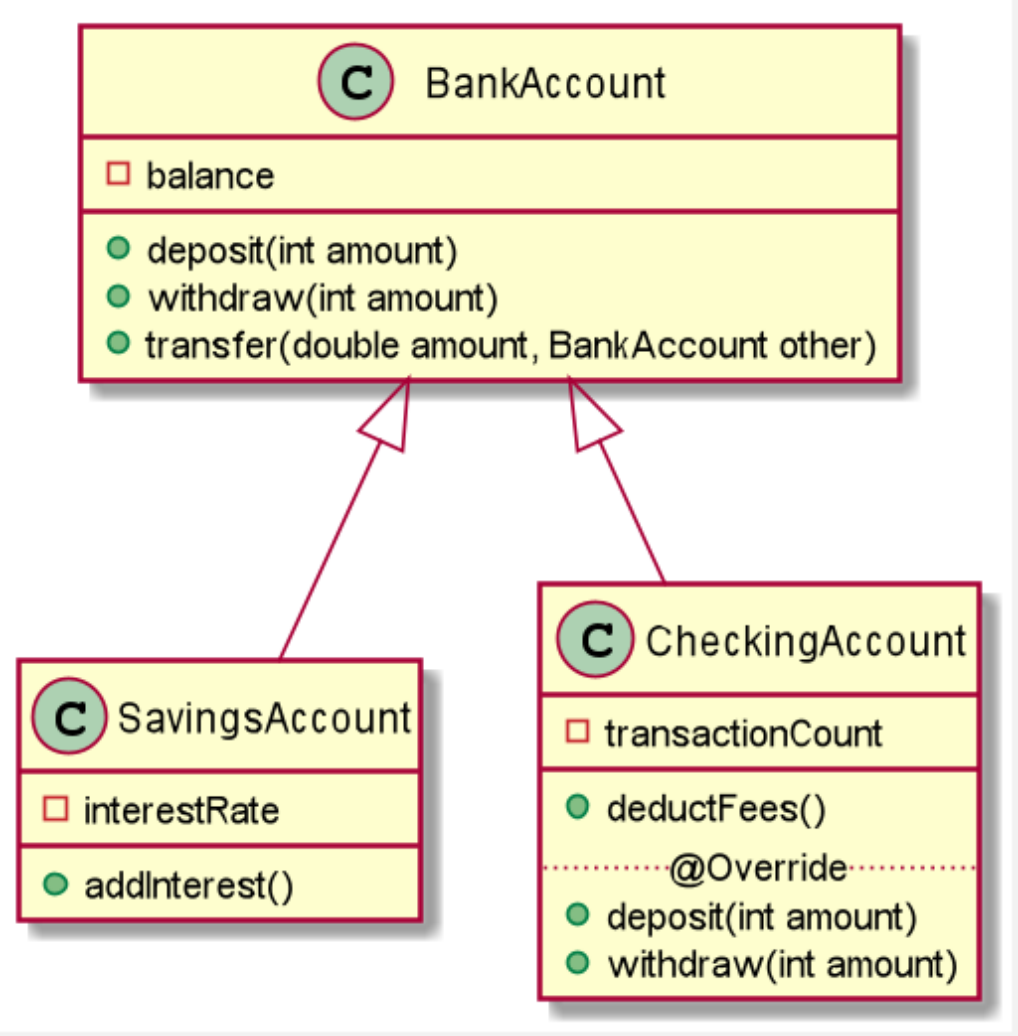


Figure 3: UML of Class relationships for BankAccount.

Summary of functionality:

The `BankAccount` class maintains a balance and allows you to deposit, withdraw and transfer to other accounts.

A `SavingsAccount` can do everything a normal bankaccount can (via inheritance) but can also accumulate interest.

A `CheckingAccount` allows us – the bank – to charge fees on the account. **Note that we've overridden `deposit()` and `withdraw()`** because we require extra functionality, i.e. we need to record that a transaction has taken place.

BankAccount Class

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{

   private double balance;

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      if (balance >= amount)
      {
           balance = balance - amount;
      }
      else
      {
         System.out.println("Withdrawal error: insufficent funds");
      }
   }
```

```
/**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }

   /**
      Transfers money from this bank account to another account
      @param amount the amount to transfer
      @param other the other account
   */
   public void transfer(double amount, BankAccount other)
   {
         //EXERCISE: Have a go at writing this.
         //HINTS:
         //It can be done in two lines of code (the basic idea, at least.)
         //Use methods - don't access balance directly.
   }

}
```

SavingsAccount class

```
/**
   An account that earns interest at a fixed rate.
*/
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   //private double balance; //never do this

   /**
    Constructs a bank account with a given interest rate.
    @param rate the interest rate (represents a percentage)
    */
   public SavingsAccount(double rate)
   {
      super();
      interestRate = rate;
   }

   /**
    *
    * @param rate the interest rate to be appiled via addInterest()
    * @param initBalance the initial balance the account opens with
    */
   public SavingsAccount(double rate, double initBalance)
   {
      super(initBalance);
      interestRate = rate;
   }

   /**
      Adds the earned interest to the account balance.
   */
   //Note the use of inherited methods to access balance indirectly
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;

      //balance += interest; //NB: Can't do this due to balance being private

      deposit(interest);
   }

}
```

CheckingAccount Class

```
/**
   A checking account that charges transaction fees.
*/
public class CheckingAccount extends BankAccount
{
   private int transactionCount;

   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;

   /**
      Constructs a checking account with a given balance.
      @param initialBalance the initial balance
   */
   public CheckingAccount(double initialBalance)
   {
      // Construct superclass
      super(initialBalance);

      // Initialize transaction count
      transactionCount = 0;
   }

   //EXERCISE
   //Fill in the missing code for deposit() and withdraw()
   //Again, the basic idea can be accomplished in both using
   //two lines of code:
   //Record that a transaction has taken place
   //Carry out the deposit/withdraw
   @Override
   public void deposit(double amount)
   {
      //complete this
   }

   @Override
   public void withdraw(double amount)
   {
      //complete this
   }

   /**
      Deducts the accumulated fees and resets the
      transaction count.
   */
   public void deductFees()
   {
      //complete this
      //Figure out if there are chargeable transactions
      //Calculate the fee and apply
      //Reset transaction count
   }
}
```

Now that we've seen the code – and hopefully, understood most of it – let's talk a little about it. Some of the material below has already been mentioned but there's no harm in a little repetition.

Basic Usage:

1. When we create a new subclass, our new class inherits all the instance-fields (attributes) and methods (behaviours) that are defined in the superclass.
   For example, the SavingsAccount class automatically inherits `balance`, `getBalance()`, `deposit()`, `withdraw()`.

2. Even though we inherit balance (and therefore our SavingsAccount objects will store two pieces of information: a balance and an interest rate), **we do not have direct access to it** in our SavingsAccount code.
   The reason for this is that it (balance) is declared private in the superclass. NOTE: you should **NEVER** change the access to public just to get round this issue.
   If you do then you've messed up the power of objects – i.e. the only code that can mess-up a class's data (instance fields) is the class's code (methods). That means that if a class is well written and tested we know that it works as a self-contained unit.
   If we suddenly start making instance-fields public then anyone can now mess-up the data, so the encapsulation provided by our classes (to our objects) is pretty much gone.

   So, we know that we do not have direct access to the superclass data, but that's not a problem if the superclass gives us indirect access to it via appropriate methods.

   In the SavingsAccount example this is provided via `getBalance()` – which is an accessor method – and `deposit()` and `withdraw()`, which are mutator methods.
   See the example below, particularly the `addInterest()` method.

```
/**
   An account that earns interest at a fixed rate.
*/
public class SavingsAccount extends BankAccount
{
   private double interestRate;

   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }

   //Note the use of inherited methods to access balance indirectly
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      //balance += interest; //NB: Can't do this due to balance being private
      deposit(interest);
   }
}
```

3. Leading on from the example in 2. above, **don't be tempted to declare a variable called `balance` in SavingsAccount just because you can't get direct access to it**.

   Unfortunately, the compiler will allow you to do this but what happens then is that if you create a SavingsAccount object **your object will actually have two balance instance fields** – one contained in the BankAccount portion and one contained in the SavingsAccount portion.

   This is called "shadowing of instance fields" - depending on what method you invoke you may be accessing/modifying a different balance each time. An account with two balances is going to be problematic!!!

Constructor Chaining:

The superclass has a constructor (or constructors) and the subclass also has one or more constructors.

If we decide to create a subclass object we must take in all information (info to initialise our instance fields) that are required to initialise that object via the subclass constructor.

In the subclass constructor the first thing we normally so is *chain* to the superclass constructor (that constructor may also end up chaining information to its superclass, if it has one).

This all sounds complex, but the reality is fairly simple, especially when you've done it once or twice.

Let's look at the code for a SavingsAccount constructor:

```
public SavingsAccount(double rate, double initBalance)
{
   super(initBalance);
   interestRate = rate;
}
```

- A SavingsAccount object holds two pieces of data – balance and interestRate.

- Therefore we have a constructor which allows us to set up both when we create our object, e.g. `SavingsAccount s1 = new SavingsAccount (10, 500.60);`

- Of course, the superclass is in charge of the balance (we don't even have access to it because it is private) so we use the keyword `super` to *chain* our SavingsAccount constructor to the BankAccount constructor. And that's it, simple!

- Just note that the call to `super()` has to be the **first** statement in your subclass constructor.
  Why?
  Well you can think of the superclass as a foundation that you're building upon.
  The foundation must be in place before you can do anything else.
  The superclass must be fully constructed before you can construct the subclass portion of the object.

Method Overriding (a basic requirement of polymorphism)

We'll need this concept for another very important concept called polymorphism. (which will be explained shortly)

An overridden method is simply a **method in a subclass that has the exact same signature as a method in the superclass**. Note that this implies it is specifically to do with inheritance.

Here's a trivial example:

```
public class SuperClass
{
   public String someMethod(int var1, double var2)
   {
      //some code in here
   }
}

public class SubClass extends SuperClass
{
   public String someMethod(int var1, double var2)
   {
      //code in here (different to the code in SuperClass, of course)
   }
}
```

`someMethod()` in `Subclass` **overrides** `someMethod()` in `SuperClass` because the signature (which is the method name, method return type, and parameter types) exactly matches.
The code in each method can – and should – be different.

Please note: overriding is different from overloading - I'll leave it to you as an exercise to do a little research to find out what method-overloading is (if you don't already know).

Why would we ever bother overriding?

Polymorphism is the short answer. Before we get to that, though, let's think about our BankAccount example. **(Note: The following section talks about the deposit() and withdraw() methods in CheckingAccount. Please at least attempt the exercise related to this (given in the code listing) before continuing to read.)**

If we have a BankAccount object we will want to deposit money so we have a `deposit()` method to allow us to do that: the signature is `public void deposit(double amt)`.

If we have a CheckingAccount object we also want to want to deposit money. That seems okay because we inherit the `deposit()` method. However, that method doesn't do enough in terms of a CheckingAccount because all it does is update the balance; it never records that a transaction has taken place.

So deposit() for CheckingAccount should do two things: Record that a transaction has taken place AND update the balance.

To provide the additional functionality we provide an additional `deposit()` in CheckAccount, i.e. we **override. Note:** we could call our method `depositIntoCheckingAccount()`, for example, but that will just make life more difficult for people who want to use our classes – A CheckingAccount IS-A BankAccount so why make people remember two different `deposit()` method names (apart from all that we won't be taking advantage of some powerful object oriented features if we start messing around like this).

Look at the difference between the two methods:

```
//FROM BankAccount
public void deposit(double amount)
{
    balance = balance + amount;
}
```

```
//FROM CheckingAccount
public void deposit(double amount)
{
    transactionCount++;
    // Now add amount to balance
    super.deposit(amount);
}
```

See how we use the superclass's deposit() to complete the operation in the CheckingAccount class.

Note that the `super.` is absolutely vital. If we just called `deposit()` it would actually call itself… infinitely!

Superclass references can refer to subclass objects (the second requirement for polymorphism)

This means, for example, that a BankAccount reference could end up referring to

- A BankAccount object
- A SavingsAccount object
- A CheckingAccount object

Here's a trivial example:

```
//normal everyday use - the reference type
//corresponds to the actual object references
BankAccount b1 = new BankAccount(1000);

//This is also possible. We declare a reference
//which ends up referring to a subclass object
BankAccount b2 = new SavingsAccount(5.0, 1000);
```

You must note, though, that if a superclass reference does actually refer to a subclass object then we can only call methods defined in the superclass via this reference. To illustrate, I will continue with the example above:

```
//This is fine since deposit() is declared in the BankAccount class
b2.deposit(250);

//This won't compile - addInterest() not in the superclass
b2.addInterest();
```

Okay, so that is an obvious limitation – **even though we are actually referencing a SavingsAccount, we can't access the full capabilities of it.**

In fact, it looks a bit useless at this stage. However, there is a way round this limitation which we will come back to. Before that, though, there is one aspect of this which turns out to be extremely useful. Here it is:

A superclass reference will call the correct overridden method (assuming, of course, that overriding is in place) depending on what type of object it is actually pointing at.

Again, a trivial example just to illustrate the point:

```
//normal everyday use - the reference type
//corresponds to the actual object references
BankAccount b1 = new BankAccount(1000);

//If I call the deposit method it will update the balance
b1.deposit(120);

//Now I change my reference to refer to a subclass object
b1 = new CheckingAccount(400);

//This time it calls the deposit method defined in the subclass
//i.e. the balance gets updated AND the transaction is noted via a counter
b1.deposit(334.50);
```

So, because there are multiple deposit() methods in the inheritance hierarchy, the one that is called depends not on the type of the reference (BankAccount) but on the object that is actually being referenced at that point in time. **That is Polymorphism**[1].

Polymorphism – A Definition

The word 'polymorphism' literally means 'a state of having many shapes' or 'the capacity to take on different forms'. When applied to object oriented programming languages like Java, it describes a language's ability to process objects of various types and classes through a single, uniform interface[2].

Please Note: Polymorphism in Java has two types: Compile-time polymorphism (static binding) and **Runtime polymorphism (dynamic binding)**. Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.

However, polymorphism – in its most general sense – normally refers to run-time polymorphism and that is the convention we'll use here also.

And I'm going to leave it there for this document. Of course, you don't yet know enough to appreciate, understand or utilise Polymorphism, so don't worry if the concept is vague to you at the moment. We'll have a proper look at it soon.

Also, there are one or two other things to note about Inheritance, but we have covered most of the main ideas here. I'll explain anything else of importance in due course.

---

[1] In the code example given, what do you think will happen to the first object (the BankAccount object with balance of 1000)?
Well, b1 no longer references it. If nothing references it then it is now effectively useless to the program as it no longer has any way to access it.
In this situation, a background process called the garbage collector will see that the object is no longer useful and will free up the memory associated with the object (so that the program will have more useful memory to utilise).
Even though this is a footnote, it is an important concept to be aware of.

Appendix A – Student and Person classes

```
/**
 * SimpleDate allows us to store date of birth info
 * I used it as an example to illustrate COMPOSITION, i.e. a Person
HAS-A d.o.b.
 *
 */
public class SimpleDate {
    private int day;
    private int month;
    private int year;

    public SimpleDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    @Override
    public String toString() {
        return "SimpleDate{" +
                "day=" + day +
                ", month=" + month +
                ", year=" + year +
                '}';
    }
}
```

```
public class Person {
    private String firstName;
    private String surName;
    //Below is an example of Composition. This means that objects
    //can be composed of other objects (in fact, firstName and
    //lastName are also examples, but not quite as obvious, perhaps
    private SimpleDate dateOfBirth;

    public Person(String firstName, String surName, SimpleDate
dateOfBirth) {
        this.firstName = firstName;
        this.surName = surName;
        this.dateOfBirth = dateOfBirth;
    }

    @Override
    public String toString() {
        return "Person{" +
                "firstName='" + firstName + '\'' +
                ", surName='" + surName + '\'' +
                ", dateOfBirth=" + dateOfBirth +
                '}';
    }
}
```

```java
//Student (the child class or subclass) INHERITS from
//Person (the parent class or superclass) using the keyword extends
public class Student extends Person { //Student IS-A Person
    private int studentId;

    public Student(String firstName, String surName, SimpleDate
dateOfBirth, int studentId) {
        super(firstName, surName, dateOfBirth);
        this.studentId = studentId;
    }

    @Override
    public String toString() {
        return "Student{" + super.toString() +
                "studentId=" + studentId +
                '}';
    }
}
```

```java
public class InheritanceTester1 {
    public static void main(String[] args)
    {
        Person p1 = new Person("Ann","Smith", new SimpleDate(22,02,1990));
        System.out.println(p1);

        //In this example I create an explicit object reference to a
        //SimpleDate object - compare with p1's creation
        SimpleDate jimDob = new SimpleDate(14,02, 1999);
        Student s1 = new Student("Jim", "Jones", jimDob, 9452);
        System.out.println(s1);
    }
}
```