

---

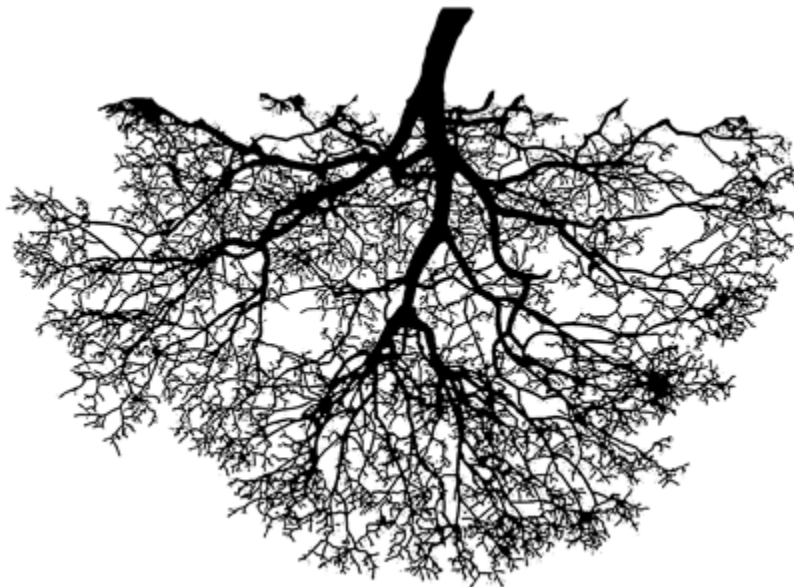
## Chapter 7

# Organizing Classes to Use Inheritance

---

## What is in This Chapter ?

Until now, we have been making simple objects that interact together to form an application. Some of the more important concepts in Object-Oriented programming arise only when we have many classes in our application, some classes being specializations of others. We will discuss here how we can organize our classes and take advantage of the notion of **Inheritance** which allows much of our code to be "shared" between similar classes called **subclasses**. The proper use of inheritance will allow us to reduce the amount of code that we need to write, resulting in quicker implementation time, less maintenance time and hence reduced costs overall. We will discuss **abstract** vs. **concrete** classes as a means of forcing users of our classes to be specific with respect to the kinds of objects that they use. Also, we will discuss **interfaces** in JAVA as a means of forcing certain classes to have particular behavior.

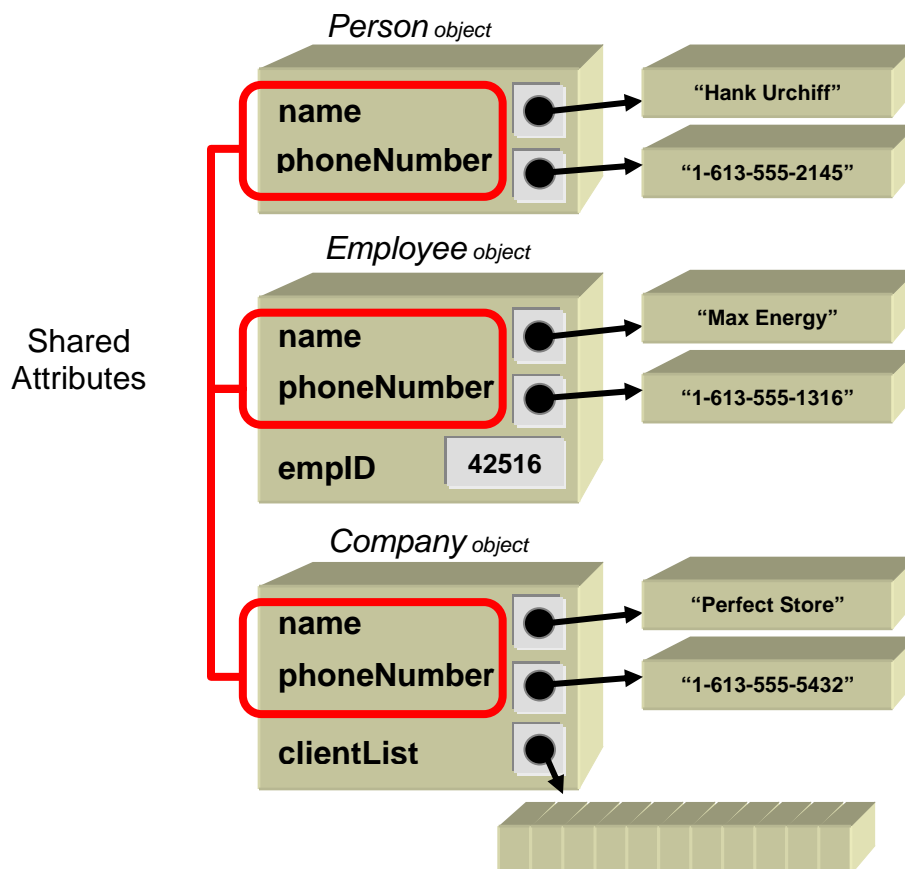


## 7.1 Class Hierarchy

We have been defining a few objects throughout the course (i.e., **Person**, **Address**, **BankAccount**, **Team**, **League**, **Car**, **Autoshow**, etc.). We created/defined a **class** for each of these objects. In fact, a definition of the word ‘**class**’ in English is: "A collection of things sharing a common attribute".

So, for example, when we created a **Person** class, we implied that all **Person** objects have some attributes in common. Similarly, a **Car** class was created to define the common attributes that **Car** objects have. In general, since **Person** and **Car** are different classes, their list of attributes will differ.

In real life, however, there are some objects that “share” attributes in common. For example, **Person** objects may have **name** and **phoneNumber** attributes, but so can **Employee**, **Manager**, **Customer** and **Company** objects. Yet, there may be attributes of these other objects that **Person** does not have. For example, an **Employee** object may maintain **empID** information or a **Company** object may have a **clientList** attribute, whereas **Person** objects in general do not keep such information.

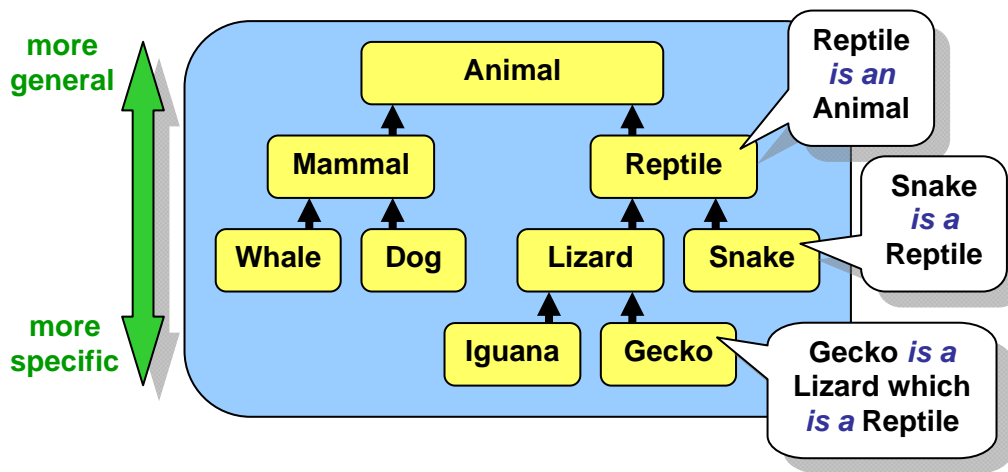
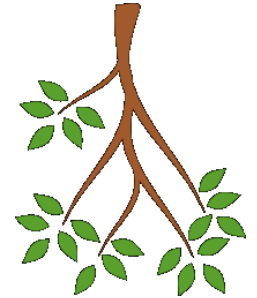


In addition to commonality between attributes, classes may also share common behavior. That is, two or more objects may have the ability to perform the same function or procedure.

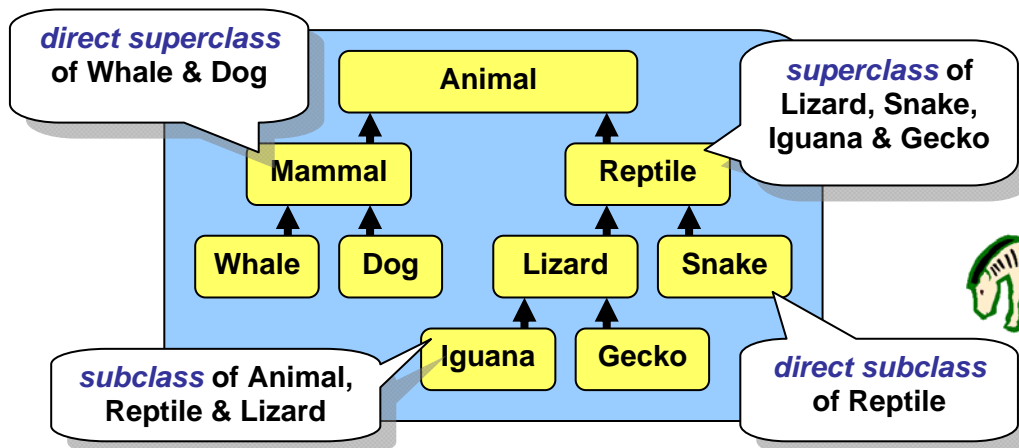
For example, if a **Person**, **Car** and **Company** are all *insurable*, then they may all have a function called **calculateInsurancePremium()** that determines the pricing information for their insurance plan.

All object-oriented languages (e.g., JAVA) allow you to organize your classes in a way that allows you to take advantage of the commonality between classes. That is, we can define a class with certain attributes (and/or behaviors) and then specify which other classes share those same attributes (and/or behaviors). As a result, we can greatly reduce the amount of duplicate code that we would be writing by not having to re-define the common attributes and/or behaviors for all of the classes that share such common features.

JAVA accomplishes this task by arranging all of its classes in a "family-tree"-like ordering called a **class hierarchy**. A class hierarchy is often represented as an upside down tree (i.e., the root of the tree at the top). The more "general" kinds of objects are higher up the tree and the more "specific" (or specialized) kinds of objects are below them in the hierarchy. So, a **child** object defined in the tree is a *more specific kind* of object than its **parent** or *ancestors* in the tree. Hence, there is an "**is a**" (i.e., "is-a-kind-of") relationship between classes:

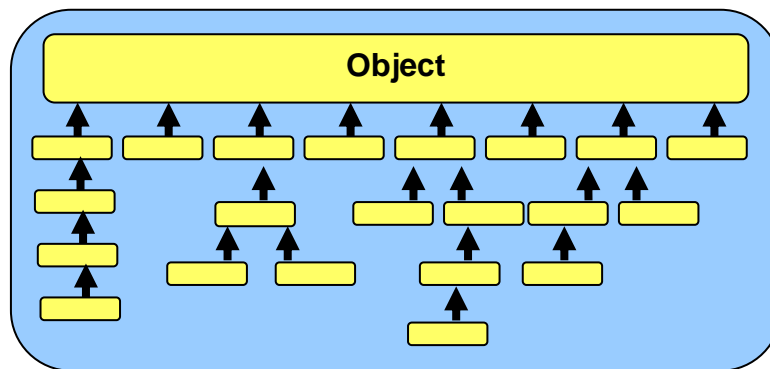


Each class is a **subclass** (i.e., a specialization) of some other class which is called its **superclass** (i.e., a generalization). The **direct superclass** is the class right "above" it:



Here, **Snake**, and **Lizard** are subclasses of **Reptile** (i.e., they are special kinds of reptiles). Also **Whale** and **Dog** are subclasses of **Mammal**. All of the classes are subclasses of **Animal** (except **Animal** itself). **Animal** is a superclass of all the classes below it, and **Mammal** is a superclass of **Whale** and **Dog**. As we can see, we can go even deeper in the hierarchy by creating subclasses of **Lizard**. Usually, when we use the term *superclass*, we are referring to the class that is directly above a particular class (i.e., the direct superclass).

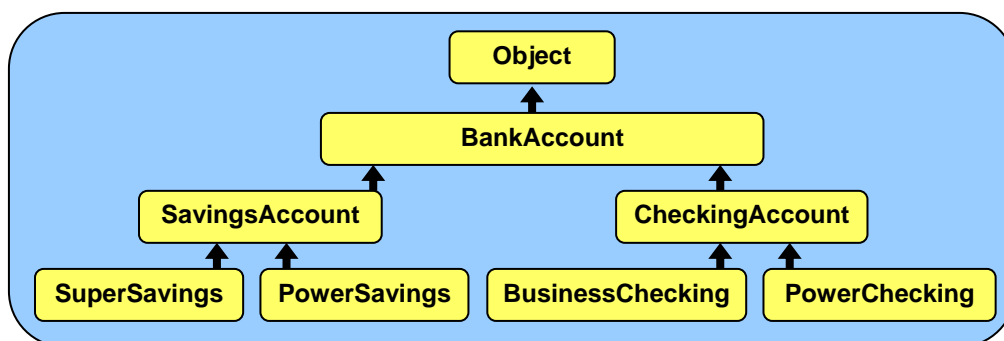
The **Animal** hierarchy above represents a set of classes that we may define ourselves. But where do they fit-in with all the other pre-made JAVA classes like **String**, **Date**, **ArrayList** etc... ? Well, all objects have one thing in common ... they are all *Objects*. Hence, at the very top of the hierarchy is a class called **Object**. Therefore, all classes in JAVA are *subclasses* of **Object**:



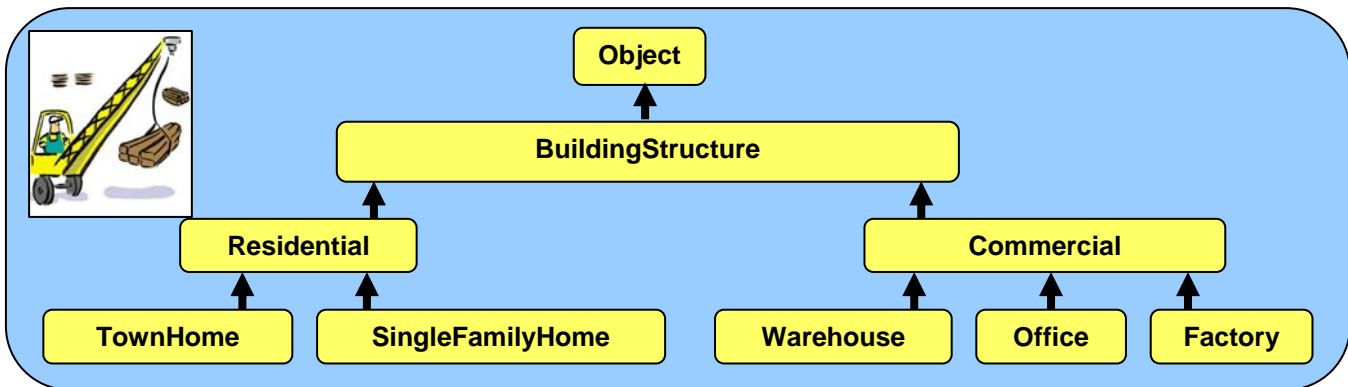
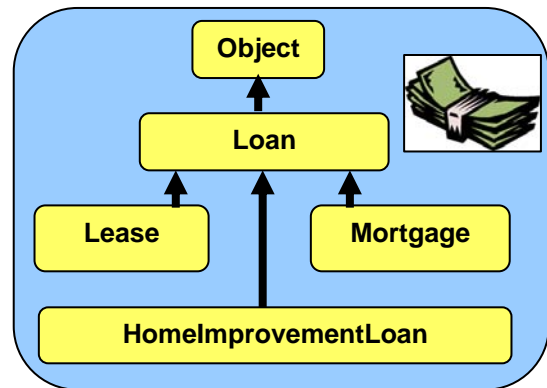
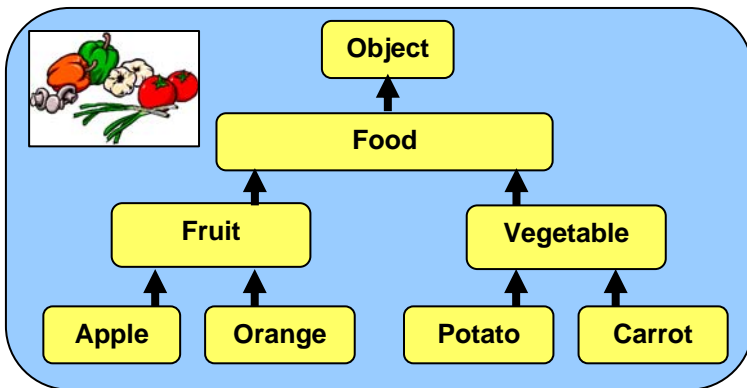
All of the classes that we created so far have been *direct* subclasses of **Object**. That means that they did not share attributes with one another, but that they shared attributes only with **Object**. However, we have the freedom to re-arrange our classes in a manner that will allow them to share attributes with one other.

The way in which we arrange our classes will depend on *how similar* our objects are with respect to their attributes. For example, a **Car** and a **Truck** have something in common ... they are both *drivable*. Whereas an **MP3Player** and a **BankAccount** have little or nothing in common with **Car** or **Truck** objects. So, intuitively, **Car** and **Truck** classes should somehow be grouped together (i.e., placed nearby) in the hierarchy.

As an example, consider creating many kinds of bank accounts. We might arrange them in a hierarchy like this:



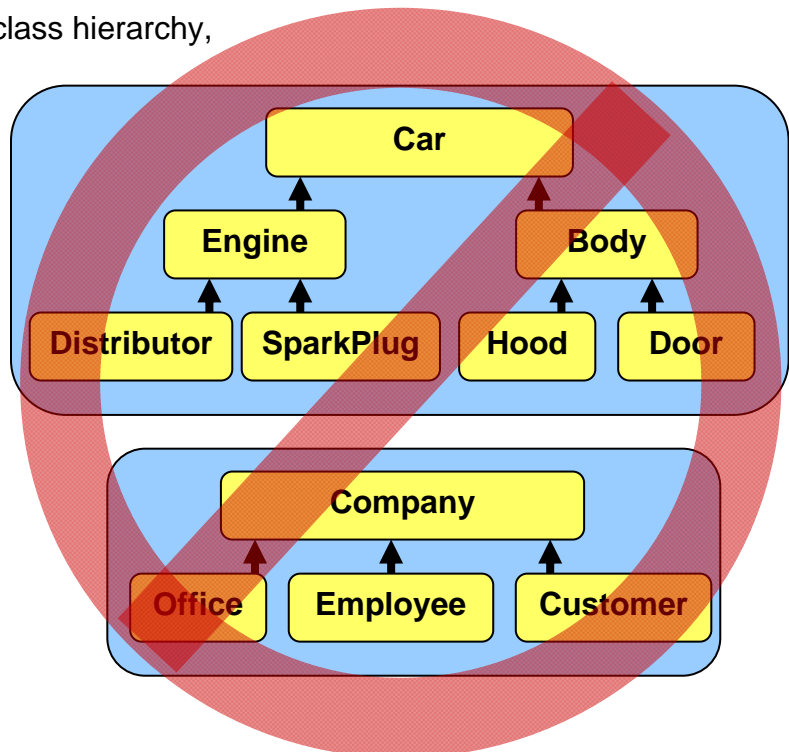
Here are a few more examples of hierarchies of classes that we may create:



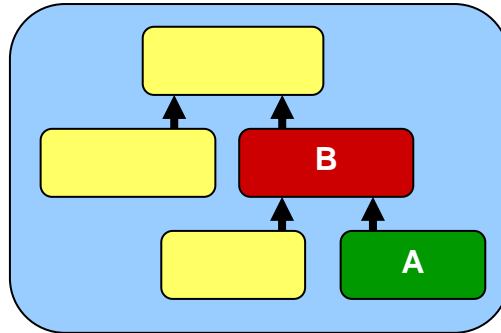
We will talk more about **how** and **why** we arrange these classes as above. But remember, a class should only be a subclass of another class if it "is a kind of" its superclass.

Sometimes, students misunderstand the class hierarchy, thinking that a class becomes a subclass of another one if the superclass "is made of" the subclasses.

That is, they mistakenly assume that it is a "has a" relationship instead of an "is a" relationship. Therefore, the following hierarchies would be wrong →



In JAVA, in order to create a subclass of another class, use the **extends** keyword in our class definition. For example, assume that we wanted to ensure that class **A** was placed in the hierarchy as a subclass of class **B** as follows:



To make this happen, we simply write **extends B** immediately after we specify name of class **A** as follows:

```
class A extends B {
    ...
}
```

If the **extends** keyword is not used (i.e., as we left it out from all our previous class definitions), it is assumed that the class being defined extends the **Object** class. So, all the classes that we defined previously were direct subclasses of **Object**.

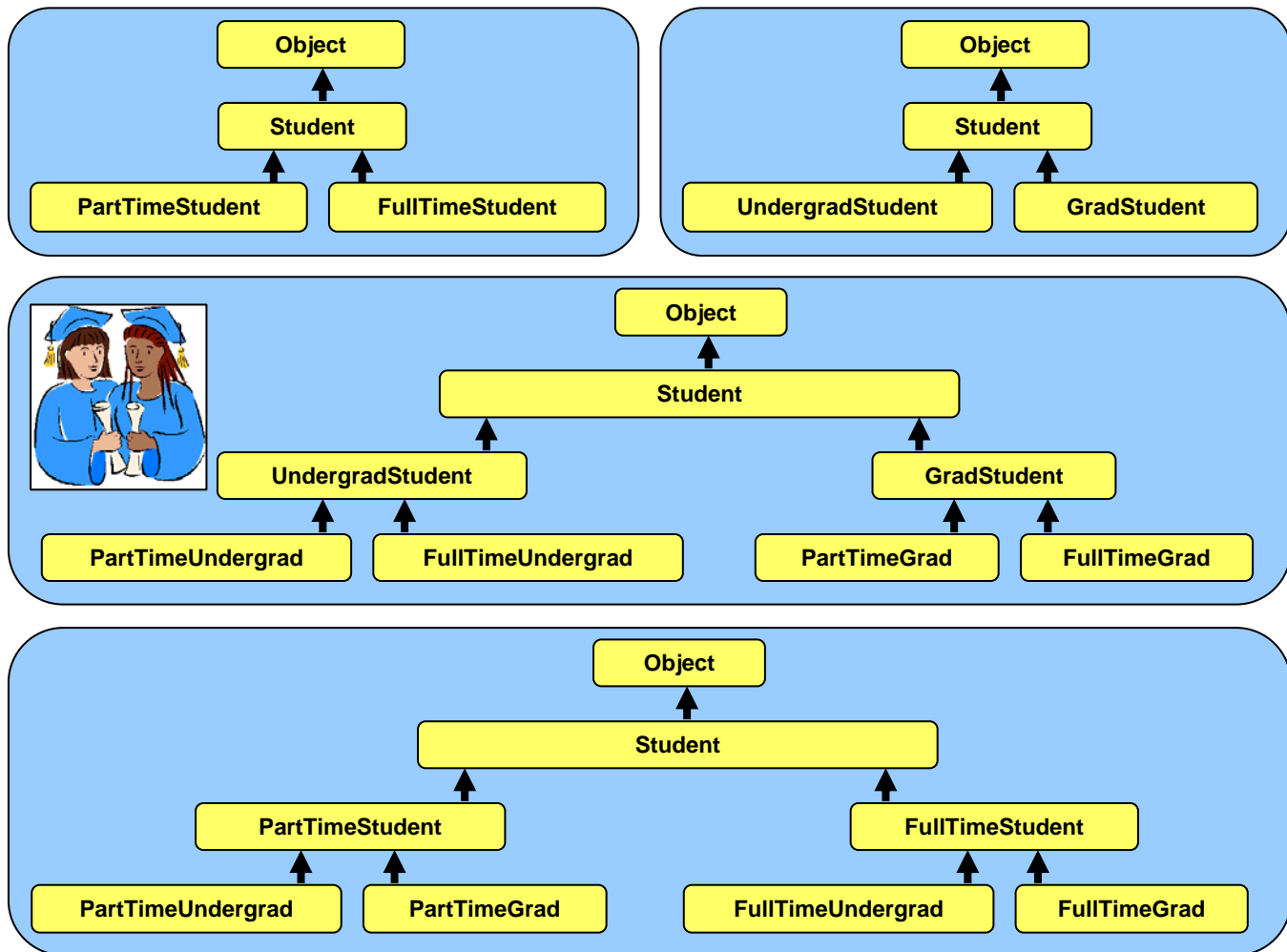
How do we know how deep we should make the class hierarchy (i.e., tree) ?

Most of the time, any “**is a**” relationship between objects should certainly result in subclassing. Object-oriented code usually involves a lot of small classes as opposed to a few large ones.



It is often the case that our class hierarchies become rearranged over time, because we often make mistakes in deciding where to place the classes. We make such mistakes because it is not always easy to choose a hierarchy ... it depends on the application.

For example, hierarchies of classes representing students in a university may be arranged in many different ways ... here are just 3 possibilities ...



How do we know which one to use? It will depend on the state (i.e., attributes) and behavior (i.e., methods) that is common between the subclasses. If we find that the main differences in attributes or behavior are between full time and part time students (e.g., fee payment rules), then we may choose the top hierarchy. If however the main differences are between graduate and undergraduate (e.g., privileges, requirements, exam styles etc.), then we may choose the middle hierarchy. The bottom hierarchy further distinguishes between full and part time graduate and undergraduate students, if that needs to be done. So ... the answer is ... we often **do not know** which hierarchy to choose until we thought about which hierarchy allows the maximum sharing of code.

The **getClass()** method in the **Object** class can be sent to any object. It returns a special **Class** object that represents the class that an object actually belongs to. We can even use the **getName()** method on a **Class** object to get a string representing the class name. Here is an example of how to use these ...

```

BankAccount    a;
Class           c;

// Make a BankAccount object and get its class
a = new BankAccount();
c = a.getClass();
System.out.println(c);           // prints class BankAccount
System.out.println(c.getName()); // prints BankAccount

```

Keep in mind that the **getClass()** method can be sent to any object ... while the **getName()** method can ONLY be sent to a **Class** object.

Similarly, we can use the **instanceof** keyword to determine whether or not an object belongs to a particular class:

```

BankAccount    a;

a = new BankAccount();
if (a instanceof BankAccount) { // same as a.getClass() == BankAccount
    ...
}

```

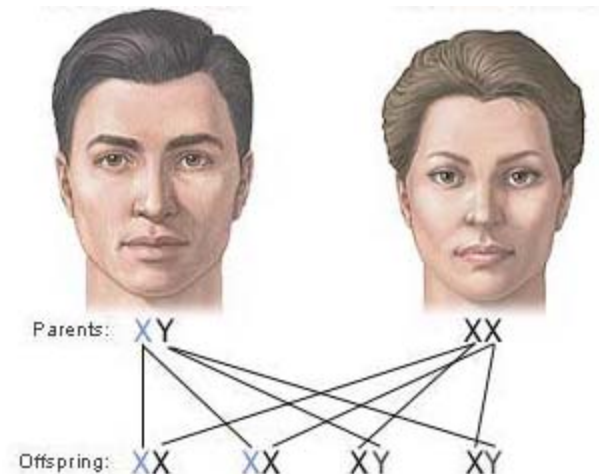
As we will see later in this section, the **instanceof** keyword can be quite useful when you need to figure out what kind of object you have (for example when you pull arbitrary objects out of a “collection” and need to make a decision based on its type).

## 7.2 Inheriting Attributes

You may have heard the term *inherit* before which has various meanings in English such as:

- “to receive from a predecessor” or
- “to receive by genetic transmission”

Through birth, all of us have *inherited traits* and **behaviors** from our parents. Something similar happens in JAVA with regards to the class hierarchy. A subclass (i.e., child) **inherits** the attributes (i.e., instance variables) and behavior (i.e., methods) from all of its superclasses (i.e., ancestors in the class hierarchy). So as a general definition, in Object-Oriented Programming, **Inheritance** is the act of receiving shared **attributes** and **behavior** from more general types of objects up the hierarchy.





This means that a subclass has the same "general" attributes/behaviors as its superclasses as well as possibly some new additional attributes/behaviors which are specific for the subclass.

There are many advantages of using **Inheritance**:

- allows code to be **shared** between classes. This promotes software re-usability
- **saves programming time** since code is shared ... less code needs to be written
- helps keep **code simple** since inheritance is natural in real life



Some languages (e.g., C++) allow **Multiple Inheritance**, which means that a class can inherit state and behavior from more than one class. However, JAVA does not support multiple inheritance. We can however, partially "fake" it (with respect to methods) through the use of *interfaces* (which we will discuss later).

Consider making an object to represent an **Employee** in a company which maintains: **name**, **address**, **phoneNumber**, **employeeNumber** and **hourlyPay**. We may make a single class for this:

```
class Employee {
    String    name;
    Address   address;
    String    phoneNumber;
    int       employeeNumber;
    float     hourlyPay;
    ...
}
```



Employee

Assume now that we have many employees in a company in which a few of them are managers. If the managers are all essentially the same as employees, except perhaps that they have a higher **hourlyPay**, then there is no need to create any new classes. The **Employee** class is sufficient to represent them.

However, what if there were some more significant differences between managers and employees? Perhaps it would be beneficial to create a separate class for them. We would need to determine **what is different** between these two classes with respect to their attributes and behaviors. For example, a **Manager** may have:

- *additional* attributes (e.g., a list of **duties**, a list of **employees** that work for them, etc...)
- *additional* (or different) behavior (e.g., they may compute their pay differently, or have different benefit packages, etc...)




In these situations, a **Manager** may be considered as a special “kind of” **Employee**. It would therefore make sense for the **Manager** to be a **subclass** of **Employee** as follows:

```

class Employee {
    String    name;
    Address   address;
    String    phoneNumber;
    int       employeeNumber;
    float     hourlyPay;
    ...
}

class Manager extends Employee {
    ArrayList<String>    duties;
    ArrayList<Employee> subordinates;
    ...
}

```

Notice here that **Manager** would inherit all of the attributes of the **Employee** class, so that **Employees** have 5 attributes, while **Managers** have 7. All **Employee** behaviors would also be inherited by **Managers**.

Now, what if we wanted to represent a **Customer** as well in our application? Our application may require keeping track of a customer’s **name**, **address** and **phoneNumber**. But these attributes are also being used for our **Employee** objects. We could make two separate unrelated classes ... one called **Customer** ... the other called **Employee**. We could define **Customer** as follows:

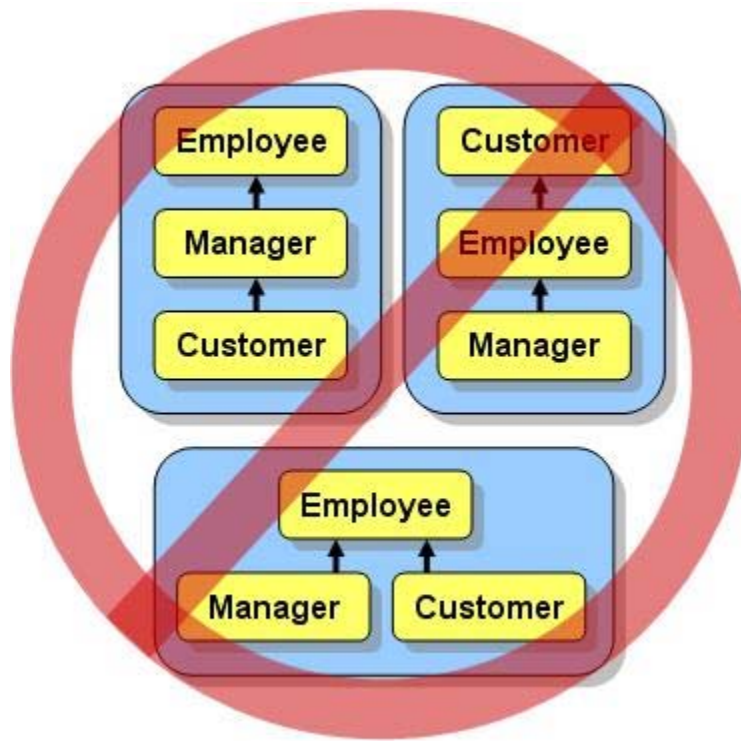
```

class Customer {
    String    name;
    Address   address;
    String    phoneNumber;
    ...
}

```



This would work fine. However, you will notice that both **Employee** and **Customer** have some attributes in common. So, if we defined the **Customer** class in this manner, we would need to repeat the same definitions, and perhaps some of the behaviors. It would be better if we could somehow use inheritance to allow **Customers** to share attributes and behaviors that are in common with **Employees**. So, we should perhaps have **Customer** inherit from something. We have a few choices. We can have **Customer** inherit from **Manager**, **Employee** inherit from **Customer** or **Customer** inherit from **Employee** as follows ...



However, neither of these hierarchies will work according to the "is a" relationship because:

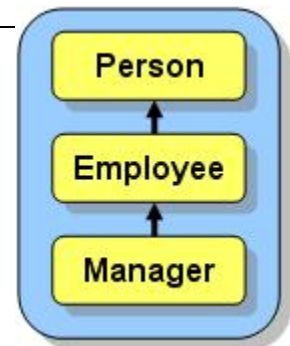
- a **Customer** is not always a **Manager**
- an **Employee** is not always a **Customer**
- a **Customer** is not always an **Employee**

One possible solution is to change the name **Customer** to **Person**. In this way, a customer is simply represented by a **Person** object and we can use the following hierarchy:

```
class Person {
    String    name;
    Address   address;
    String    phoneNumber;
    ...
}


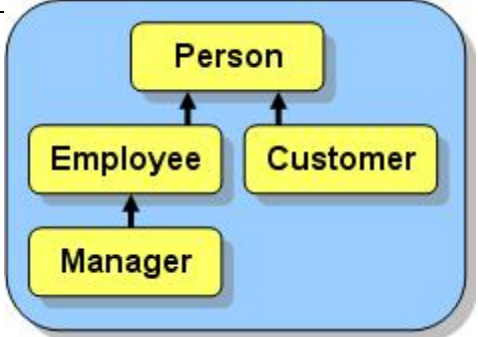



class Employee extends Person {
    int       employeeNumber;
    float     hourlyPay;
    ...
}

class Manager extends Employee {
    ArrayList<String>    duties;
    ArrayList<Employee> subordinates;
    ...
}
```



Now **Employee** inherits 3 attributes from **Person**, so it has 5 altogether, while **Manager** inherits 3 from **Person** and 2 from **Employee**, making 7 altogether. **Customers**, are then represented simply as **Person** objects.

This is a good solution as long as ALL of the attributes (e.g., **name**, **address**, **phone number**) for a customer (i.e., **Person** object) is also shared with **Employee** and **Manager**. Also, there must not be any attributes or behaviors in the **Person** class that do not apply to an **Employee** and a **Manager**. For example, if the application required us to keep track of a list of items purchased by the customer or perhaps even a purchase history, then such attributes may not make sense for an **Employee** or **Manager**. So, if there is different behavior or attributes that is unique to customers, then we must create a separate **Customer** class to define these differences. In this case, we can still share the **name**, **address** and **phoneNumber** by creating an *extra* **Customer** class to hold the common attributes. We can create the following hierarchy:

<pre>class Person {     String    name;     Address   address;     String    phoneNumber;     ... }</pre>		
<pre>class Employee extends Person {     int        employeeNumber;     float      hourlyPay;     ... }</pre>		
<pre>class Customer extends Person {     ArrayList&lt;String&gt; itemsPurchased;     ArrayList&lt;Date&gt;   purchaseHistory;     ... }</pre>		
<pre>class Manager extends Employee {     ArrayList&lt;String&gt; duties;     ArrayList&lt;Employee&gt; subordinates;     ... }</pre>		

This will allow all common attributes (i.e., **name**, **address**, **phoneNumber**) to be shared by all the classes while allowing **Customer** objects to have their own attributes and behaviors.

At this point, we should clarify the advantages of the attribute-related inheritance that is occurring within our hierarchy. Here is a simple example piece of code showing the attributes that are readily available to each type of object defined in our example ...

```

Person    p = new Person();
Employee  e = new Employee();
Customer  c = new Customer();
Manager   m = new Manager();

p.name = "Hank Urchiff";           // own attribute
p.address = new Address();         // own attribute
p.phoneNumber = "1-613-555-2328"; // own attribute

e.name = "Minnie Mumwage";        // attribute inherited from Person
e.address = new Address();         // attribute inherited from Person
e.phoneNumber = "1-613-555-1231"; // attribute inherited from Person
e.employeeNumber = 232867;        // own attribute
e.hourlyPay = 8.75f;              // own attribute

c.name = "Jim Clothes";           // attribute inherited from Person
c.address = new Address();         // attribute inherited from Person
c.phoneNumber = "1-613-555-5675"; // attribute inherited from Person
c.itemsPurchased.add("Pencil Case"); // own attribute
c.purchaseHistory.add(Date.today()); // own attribute

m.name = "Max E. Mumwage";        // attribute inherited from Person
m.address = new Address();         // attribute inherited from Person
m.phoneNumber = "1-613-555-8732"; // attribute inherited from Person
m.employeeNumber = 232867;        // attribute inherited from Employee
m.hourlyPay = 8.75f;              // attribute inherited from Employee
m.duties.add("Phone Clients");    // own attribute
m.subordinates.add(e);            // own attribute

```

Notice that we use the inherited attributes just as if they were defined as part of that class directly. For example, the **Employee** object **e**, **Customer** object **c** and **Manager** object **m**, all access the **name** attribute as if it was defined in their class ... even though it is actually defined in the **Person** class ... written in a different **.java** file!! You can see that through inheritance, we do not have to re-define the **name** attribute in each of these classes. The same holds true for the **address** and **phoneNumber** attributes, as well as any other inherited attributes in the subclasses.

At this point, we only examined how to decide upon a class hierarchy based on the differences in attributes. However, we would have to think in the same manner by examining the behaviors of the individual classes. For example, even if managers did not have the **duties** and **subordinates** attributes shown above, we may still want to make a separate class for managers if there are behaviors that differ (e.g., different **computePay()** method). In the next section, we will consider an example that shows how inheritance applies to behaviors within a simple hierarchy of **BankAccount** objects.

## 7.3 Inheriting Behavior

Consider creating an application for a bank that maintains account information for its customers. All bank accounts at this bank must maintain 3 common attributes (the **owner's** name, the **account number** and the **balance** of money remaining in the account). Also, an account, by default, should have simple behaviors to **deposit** and **withdraw** from the account. So, in its simplest form, a **BankAccount** object can be defined and used as follows:

```
class BankAccount {
    static int LAST_ACCOUNT_NUMBER = 100000;

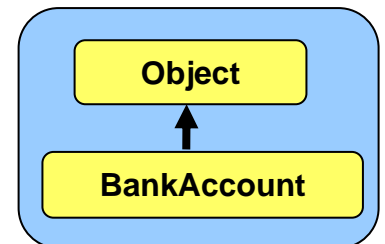
    String    owner;           // person who owns the account
    int       accountNumber;   // the account number
    float     balance;        // amount of money currently in the account

    // Some constructors
    BankAccount() {
        this.owner = "";
        this.accountNumber = LAST_ACCOUNT_NUMBER++;
        this.balance = 0;
    }
    BankAccount(String ownerName) {
        this.owner = ownerName;
        this.accountNumber = LAST_ACCOUNT_NUMBER++;
        this.balance = 0;
    }

    // Return a string representation of the account
    public String toString() {
        return "Account #" + this.accountNumber + " with $" + this.balance;
    }

    // Deposit money into the account
    void deposit(float amount) {
        this.balance += amount;
    }

    // Withdraw money from the account
    void withdraw(float amount) {
        if (this.balance >= amount)
            this.balance -= amount;
    }
}
```



Now assume that the bank wants to distinguish between “*savings*” accounts and “*non-savings*” accounts in that the customer cannot withdraw money from a “*savings*” account once it has been deposited (i.e., to get the money out of the account, the customer must close the account).



We would need to have a way of disabling the withdraw behavior for *savings* accounts. We could do this through inheritance by creating a subclass of **BankAccount** to represent a special “kind of” account ... we will call it **SavingsAccount**:

```
class SavingsAccount extends BankAccount {
}
```

Just by writing this simple “virtually empty” class definition in which **SavingsAccount** *extends* **BankAccount**, we have “invented” a new type of bank account that inherits all 3 attributes from **BankAccount** as well as the **toString()**, **deposit()** and **withdraw()** methods. We could verify this by writing a simple piece of test code:



```
SavingsAccount s = new SavingsAccount();

System.out.println(s);           // displays Account #100000 with $0.0
s.deposit(120);
System.out.println(s);           // displays Account #100000 with $120.0
s.withdraw(20);
System.out.println(s);           // displays Account #100000 with $100.0
```

Something important to know, however, is that a subclass does not automatically inherit the constructors in its superclass. So, **SavingsAccount** does not inherit the two constructors in **BankAccount** ... but it does get to use its own default constructor (i.e., zero-parameter constructor) for free. We can verify this by altering the first line in our test code so read:

```
SavingsAccount s = new SavingsAccount("Bob");
```

If we made such an alteration to the code, our test code would no longer compile. We would receive the following compile error:

```
cannot find symbol constructor SavingsAccount(java.lang.String)
```

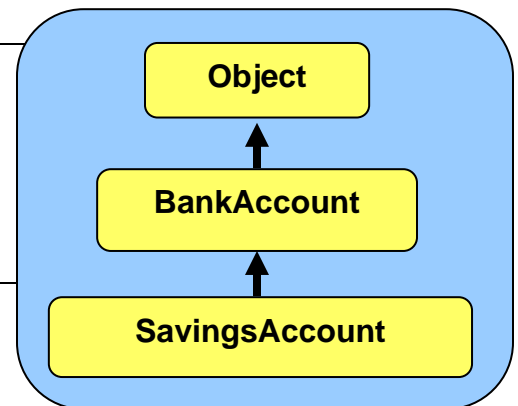
which is telling us that we don't have a constructor in our **SavingsAccount** class that takes a single **String** parameter. How then did our `new SavingsAccount()` code work previously since it seems to have properly initialized the account number? Well, as it turns out, the default constructor that we get for free actually looks as follows:

```
SavingsAccount() {
    super();
}
```

What does this mean ? What does the keyword **super** do ? The keyword **super** is actually a special word that represents the *superclass* of this class. In our case, the **super** class is **BankAccount**. So, it is essentially doing a call to **BankAccount()** ... which means it is calling the superclass constructor. Therefore, if we want to make use of the attribute initialization code that is in a constructor in a superclass, we can call the superclass constructor from our own by using **super(...)** along with the appropriate parameters. Hence we can write the following constructor in our **SavingsAccount** class:



```
class SavingsAccount extends BankAccount {
    SavingsAccount(String aName) {
        super(aName);
    }
}
```



If we do this, then we can use the following code without compile errors:

```
SavingsAccount s = new SavingsAccount("Bob");
```

Keep in mind, however, that the list of parameters (i.e., the types) supplied within the **super(...)** call, must match the list of parameters (i.e., the types) of one of the constructors in the superclass. In order to see the advantage of using constructor inheritance, here is what the code would look like with and without using inherited constructors:

Without Inheritance (need to re-write the code)	With Inheritance
<pre>SavingsAccount() { {     this.owner = "";     this.accountNumber = LAST_ACCOUNT_NUMBER++;     this.balance = 0; }  SavingsAccount(String ownerName) {     this.owner = ownerName;     this.accountNumber = LAST_ACCOUNT_NUMBER++;     this.balance = 0; }</pre>	<pre>SavingsAccount() {     super(""); }  SavingsAccount(String aName) {     super(aName); }</pre>



Again ... the amount of code that needs to be written is reduced when using inheritance.

So, we have **SavingsAccount** properly inheriting from **BankAccount**, however, the **SavingsAccount** class still allows withdrawals. In order to disable this behavior, we need to somehow “prevent” the withdraw method code from being used by savings accounts. The simplest and most common way of doing this is to write a new **withdraw()** method in the **SavingsAccount** class that simply does nothing as follows ...



```

class SavingsAccount extends BankAccount {
    // Constructor to call the superclass constructor
    SavingsAccount(String aName) { super(aName); }
    SavingsAccount() { super(""); }

    // Prevent the withdrawal of money from the account
    void withdraw(float amount) {
        // Do nothing
    }
}

```

Once we re-compile, we can test it out by running our test code again:

```

SavingsAccount s = new SavingsAccount();

System.out.println(s);           // displays Account #100000 with $0.0

s.deposit(120);
System.out.println(s);           // displays Account #100000 with $120.0

s.withdraw(20);
System.out.println(s);           // this will do nothing now
                                 // displays Account #100000 with $120.0

```

Notice that the test code remains the same but now it no longer performs the withdrawal calculation. What is actually happening here? By writing the **withdraw()** method in the **SavingsAccount** class, we are actually **overriding** the one that is in the **BankAccount** class. That is, we are replacing the inherited behavior with our own unique behavior. So, we are *preventing* or *disabling* the inheritance for this behavior.

The idea of overriding behavior is actually not new to us. Every time that we write a **toString()** method, we are actually overriding the one that we would normally inherit from the **Object** class.

At this point, we now have **SavingsAccounts** that cannot be withdrawn from and normal **BankAccounts** that can be withdrawn from. Lets see another way that we can use overriding ... to *modify* inherited behavior.

Assume that the bank also wants to encourage depositing to savings accounts by giving \$0.50 to the customer for each \$100 that they deposit into their **SavingsAccount** (i.e., not for regular **BankAccounts**). For example, if they deposit \$354.23, then their account balance should immediately increase by \$355.73 ... showing the extra \$1.50 applied to the deposit amount.



To do this, we can completely override the deposit method from **BankAccount** by writing the following method in **SavingsAccount** ...

```
// Deposit money into the account
void deposit(float amount) {
    this.balance += amount;

    // Now add the bonus 50 cents per $100
    int wholeDollars = (int)(amount/100);
    this.balance += wholeDollars * 0.50f;
}
```

This method of overriding would work fine and would properly add the extra bonus deposit incentive. However, the first line is a duplication of the **BankAccount** class's **deposit()** method. This duplication may seem insignificant in this simple example, but in a real bank application there may actually be much more code devoted to the deposit process (e.g., logging the transaction). Hence, it would be better to make use of inheritance.

How though, can we inherit the **deposit()** method in **BankAccount**, while also incorporating the additional bonus deposit behavior necessary for **SavingsAccounts**? The answer makes use of the **super** keyword again. Here is the solution:

```
// Deposit money into the account
void deposit(float amount) {
    // Call the deposit() method in the superclass
    super.deposit(amount);

    // Now add the bonus 50 cents per $100
    int wholeDollars = (int)(amount/100);
    this.balance += wholeDollars * 0.50f;
}
```



Notice that this time we use a dot **.** after the **super** keyword, followed by the method that we want to call in the superclass. Here, the word **super** is used to tell JAVA to look for the **deposit()** method in the superclass. JAVA will go and evaluate the superclass **deposit()** method (which performs the “normal” depositing process) and then return here and complete the behavior by adding the 50 cent bonus incentive. This method is still considered to *override* the **deposit()** method in **BankAccount**. It is an example of a situation in which we want to “borrow” a superclass’s behavior, but then add some additional behavior as well.

Alternatively, we could have combined the deposit amount with the 50 cent bonus incentive before calling the superclass method as follows:

```
// Deposit money into the account
void deposit(float amount) {
    int wholeDollars = (int)(amount/100);
    super.deposit(amount + (wholeDollars * 0.50f));
}
```

or even simpler:

```
// Deposit money into the account
void deposit(float amount) {
    super.deposit(amount + (int)(amount/100)* 0.50f);
}
```



I'm sure you will agree that the overriding can be quite powerful tool to save coding time.

Just so you understand ... what would happen if we used **this** instead of **super** as follows:

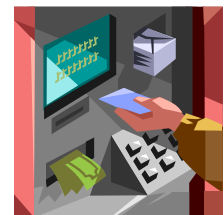
```
// Deposit money into the account
void deposit(float amount) {
    this.deposit(amount + (int)(amount/100)* 0.50f);
}
```



Well, we would be asking JAVA to call the **deposit()** method in **this** class, not the one in **BankAccount**. Furthermore, since this code is written **inside** the **deposit()** method, we are telling JAVA to call the method that we are actually trying to write! So the method will keep calling itself forever ... an infinite loop! We would get a pile of runtime error messages that says something like this:

```
Exception in thread "main" java.lang.StackOverflowError
  at SavingsAccount.deposit(SavingsAccount.java:13)
  at SavingsAccount.deposit(SavingsAccount.java:13)
  at SavingsAccount.deposit(SavingsAccount.java:13)
  ...
  at SavingsAccount.deposit(SavingsAccount.java:13)
```

OK. Now assume that the bank application needs to further distinguish between accounts in that it also has a special “power savings” account that is a special type of savings account that allows withdrawals, but there is a \$1.25 service fee each time a withdrawal is made. As before, this new type of account should also have the 50 cent incentive for each \$100 deposited.



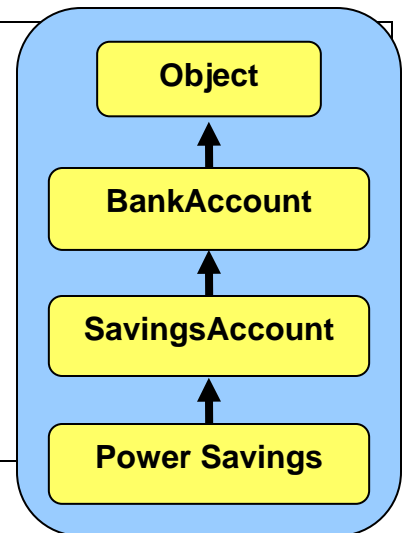
Assuming that we call the new class **PowerSavings**, where do we put it in the hierarchy? We need it to inherit the **deposit()** method from **SavingsAccount** but the **withdraw()** method from **BankAccount**. If we make **PowerSavings** a subclass of **SavingsAccount**, we will inherit the **deposit()** behavior that we want, but would then need to write a new **withdraw()** method, since the one in **SavingsAccount** does nothing. We could do this ...

```

class PowerSavings extends SavingsAccount {
    // Constructor to call the superclass constructor
    PowerSavings(String aName) { super(aName); }
    PowerSavings() { super(""); }

    // Withdraw money from the account
    void withdraw(float amount) {
        if (this.balance >= (amount + 1.50f))
            this.balance -= (amount + 1.50f);
    }
}

```



This code would work fine.

Again, we are using *overriding* by having the **withdraw()** method in **PowerSavings** override the default behavior in **SavingsAccount**. We can test our new class with the following test code:

```

PowerSavings    s = new PowerSavings();

System.out.println(s);           // displays Account #100000 with $0.0

s.deposit(320);
System.out.println(s);           // displays Account #100000 with $321.50

s.withdraw(20);
System.out.println(s);           // displays Account #100000 with $300.0

```

Notice that the **withdraw()** method properly deducts the \$1.50 fee.

However, again we are duplicating code. The code here is small, however in a large system, there may be more complicated code for withdrawing money (e.g., transaction logging, overdraft allowances, etc...). So, we do not want to duplicate this code. In fact, it would be nice if we could do something like this to call the **withdraw()** method code up in **BankAccount**:

```

class PowerSavings extends SavingsAccount {
    // Constructor to call the superclass constructor
    PowerSavings(String aName) { super(aName); }
    PowerSavings() { super(""); }

    // Withdraw money from the account
    void withdraw(float amount) {
        super.withdraw(amount + 1.50f);
    }
}

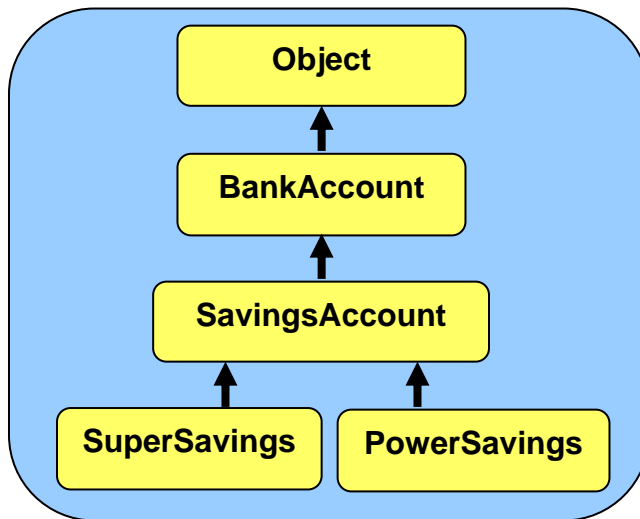
```

But this won't work. Why not? Because `super` refers to the **SavingsAccount** class here, and so it calls the **withdraw()** method in **SavingsAccount** that does nothing. In a way, what we want to do is something like this:

```
super.super.withdraw(amount + 1.50f); // super-duper does not work
```



Unfortunately, we cannot skip over a class when looking up the class hierarchy for a method. What can we do then? The solution is to re-organize our hierarchy. We seem to need common deposit behavior for savings accounts, but then differing withdrawal behavior. In reality, we actually need to distinguish between the two kinds of savings accounts. We will rename **SavingsAccount** to **SuperSavings** which will represent the previous savings account behavior. Then we will create a new **SavingsAccount** class that will contain the shared deposit behavior between the two types of savings accounts. Here is the new hierarchy:



Here is the code:

```
class SavingsAccount extends BankAccount {
    SavingsAccount(String aName) { super(aName); }
    SavingsAccount() { super(""); }

    void deposit(float amount) {
        super.deposit(amount + (int)(amount/100)* 0.50f);
    }
}
```

```
class SuperSavings extends SavingsAccount {
    SuperSavings(String aName) { super(aName); }
    SuperSavings() { super(""); }

    void withdraw(float amount) { /* Do nothing */ }
}
```

```

class PowerSavings extends SavingsAccount {
    PowerSavings(String aName) { super(aName); }
    PowerSavings() { super(""); }

    void withdraw(float amount) {
        super.withdraw(amount + 1.50f);
    }
}

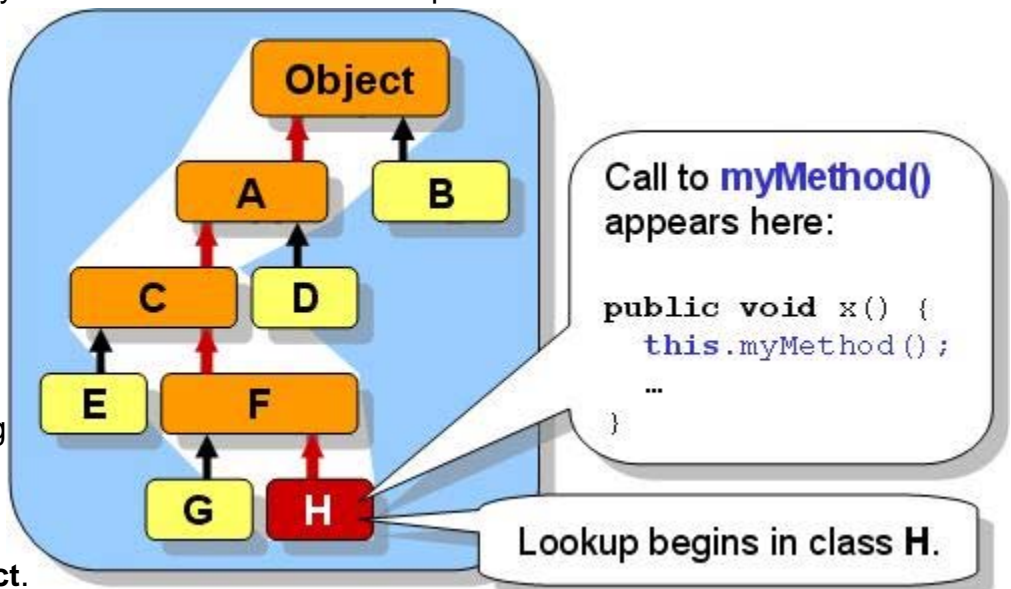
```

The code will work as we expect it to now, taking full advantage of inheritance. You may have wondered about the **toString()** method. In all of our bank account subclasses, we are able to inherit the **toString()** method that was written in the **BankAccount** class. So, for example, **PowerSavings** accounts do not inherit the **toString()** method from its direct superclass **SavingsAccount**, rather it inherits it from the **BankAccount** class further up the hierarchy. That is because of JAVA's strategy for looking up methods when you call them.

Whenever you call a method from a class directly (e.g., **this.myMethod()**), JAVA looks first to see whether or not you have such a method in the class that you are calling it from. If it finds it there, it evaluates the code in that method. Otherwise, JAVA tries to look for the method up the hierarchy (never down the hierarchy) by checking the superclass. If not found there, JAVA continues looking up the hierarchy until it either finds the method that you are trying to call, or until it reaches the **Object** class at the top of the tree.

Here is the general strategy for all instance method lookup:

- If method **myMethod()** exists in class **H**, then it is evaluated.
- Otherwise, JAVA checks the superclass of **H** for **myMethod** (in this case class **F**).
- If not found there, JAVA continues looking up the hierarchy until **Object** is reached, visiting additional classes **C**, **A** and **Object**.



If not found at all during this search up to the **Object** class, the compiler will catch this and inform you that it cannot find method **myMethod()** for the object you are trying to sending it to:

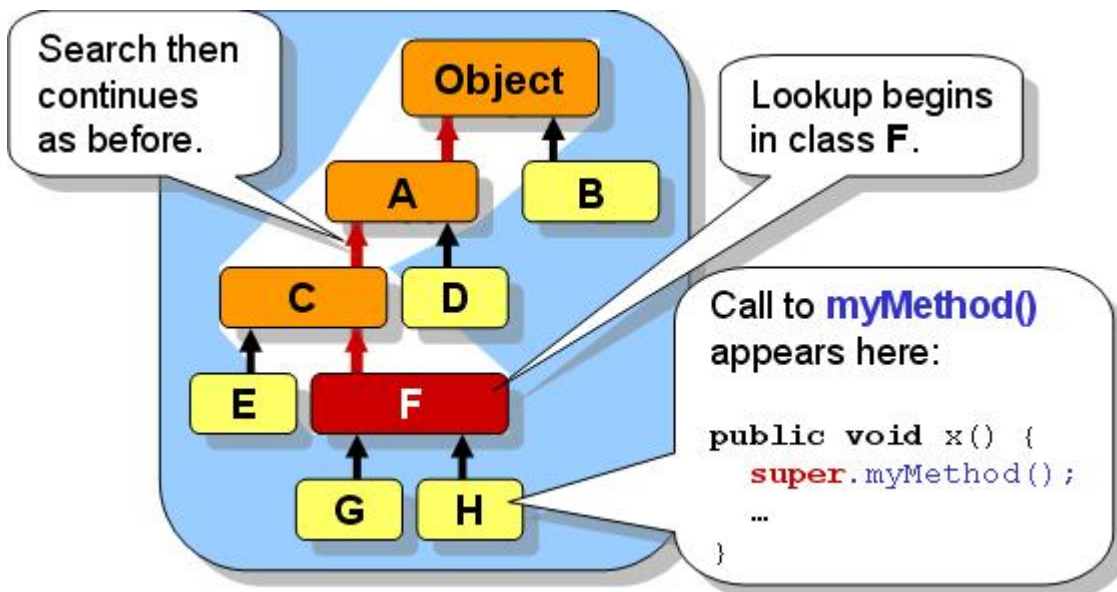
```

C:\Test.java:20: cannot resolve symbol
symbol   : method myMethod ()

```

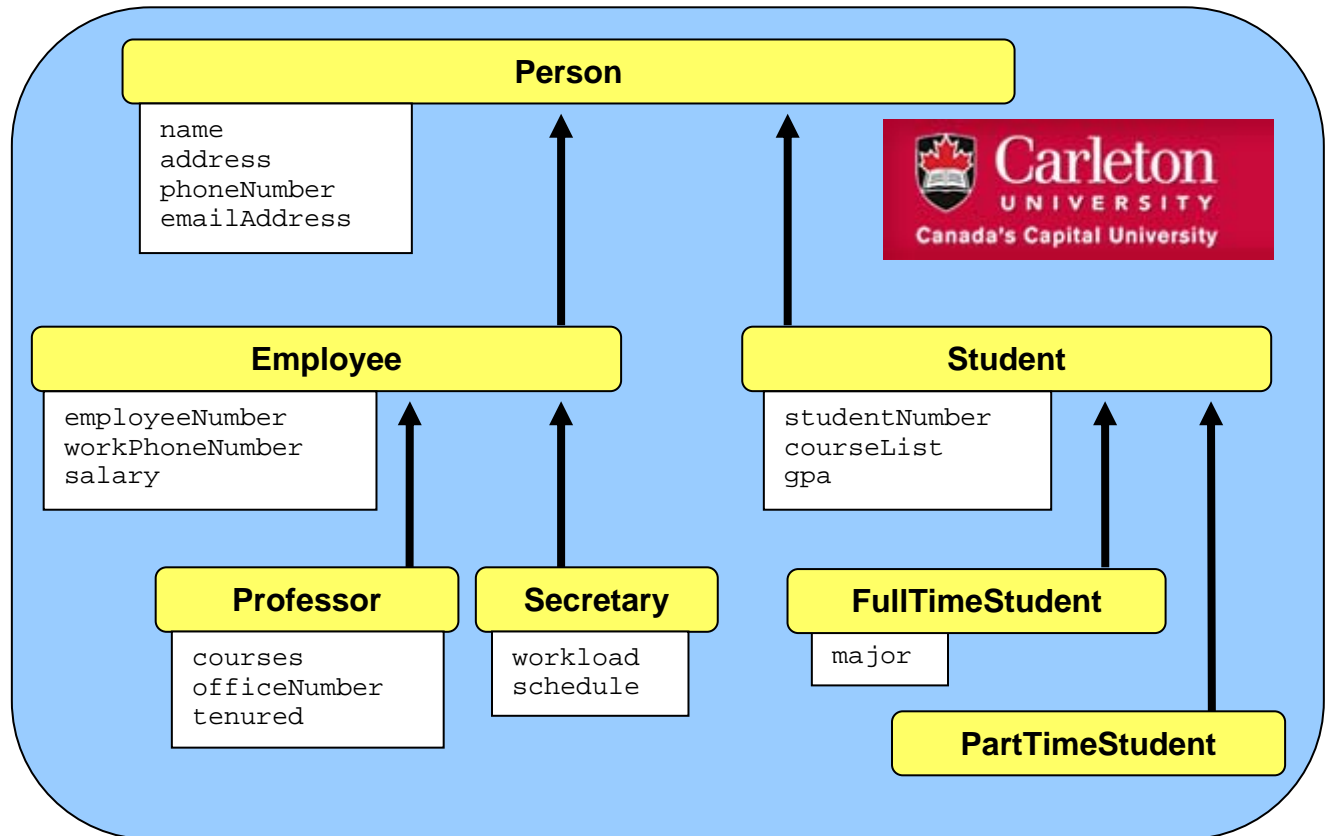
If there were many implementations of **myMethod()** along the path in the hierarchy (e.g., classes **F**, **C**, and **A** all implement **myMethod()**), then JAVA will execute the first one that it finds during its bottom-up search.

Notice the use of the keyword **this** in the picture. That tells JAVA to start looking for the method in "**this**" class. Alternatively, we can also use the keyword **super** here (i.e., **super.myMethod()**) to tell JAVA to *start* its search for the method in the *superclass*. So, if we used **super** in the example above, JAVA would start looking for **myMethod()** in class **F** first. If not found, it would then continue on up the tree looking for the method as usual. In fact if there was an implementation of **myMethod()** in the **H** class, it would not be called if we used **super**, since the search begins in the superclass, not in **this** class. So, the use of **super** merely specifies "*where the method lookup should **begin** the search*" ... nothing more.



## 7.4 University Database Example

Consider implementing a university database system. Perhaps we might come up with the following hierarchy showing the different types of people. The attributes are listed below the classes as well.



From this example we can see that there are different specializations of people but each of these special people share some common attributes (e.g., **name**, **phoneNumber** and **address**). Notice that **PartTimeStudents** do not have any of their own attributes, but will inherit 7 attributes from its superclasses **Student** and **Person**. (On a side note, do you know why would we create a subclass that does not have any additional attributes? How is a **PartTimeStudent** object different from a **Student** object? Well, **PartTimeStudents** may have different behavior as well which is implemented in that class.)

Here are the class definitions showing the attributes defined in each class ...



```
class Person {
    String name;
    String address;
    String phoneNumber;
    String emailAddress;
    ...
}
```

```
class Employee extends Person {
    int employeeNumber;
    String workPhoneNumber;
    float salary;
    ...
}
```

```
class Professor extends Employee {
    ArrayList<Course> courses;
    String officeNumber;
    boolean tenured;
    ...
}
```

```
class Secretary extends Employee {
    ArrayList<String> workLoad;
    ArrayList<WorkTask> schedule;
    ...
}
```

```
class Student extends Person {
    int studentNumber;
    ArrayList<Course> courseList;
    float gpa;
    ...
}
```

```
class FullTimeStudent extends Student {
    String major;
    ...
}
```

```
class PartTimeStudent extends Student {
    ...
}
```

Consider writing a **getDescription()** method for the **Person**, **Employee**, **Professor**, **Secretary** and **Student** classes that returns a String showing information about the object as follows:

**"Person**

```
NAME:      Jim Class
ADDRESS:   1445 Porter St.
PHONE #:   613-555-3232"
```

**"Employee**

```
NAME:      Rob Banks
ADDRESS:   789 ScotiaBank Road.
PHONE #:   613-555-2332
EMPLOYEE #: 88765
WORK #:    613-555-2433"
```

**"Professor**

```
NAME:      Guy Smart
EMPLOYEE #: 65445
WORK #:    613-555-3415
OFFICE #:   5240 PA"
```

**"Secretary**

```
NAME:      Earl E. Bird
ADDRESS:   12 Knowhere Cres.
PHONE #:   613-555-7854
EMPLOYEE #: 76845
WORK #:    613-555-3243"
```

**"FullTimeStudent**

```
NAME:      May I. Passplease
ADDRESS:   5567 Java Drive
PHONE #:   613-555-8923
STUDENT #: 100156753
MAJOR:     Computer Science"
```

**"PartTimeStudent**

```
NAME:      Al B. Back
ADDRESS:   23 Return Blvd.
PHONE #:   613-555-9738
STUDENT #: 100134257"
```

Notice that the strings span multiple lines. Also, notice that the name of the class is in the strings as well. Also note that only some of the object's attributes are shown in each case. For example, the courses were not displayed for **Professors** or **Students**. To obtain the desired results for each class, we can simply write a separate **getDescription()** method for each class that does what we want. However, this would be wasteful. Let us try to save ourselves from writing a lot of code. We will try to "share" code among classes by making use of inheritance and overriding by using the **super** keyword.

First, we notice that all classes show the **name**. This should definitely be done in the **Person** class since all others are subclasses and can inherit the code. Next, we see that all classes (except **Professor**) also show the **phoneNumber** and **address**. (I guess **Professors** do not want to show everybody this information). Thus, we will allow all subclasses to display this information and we will have to do something different for the **Professor** class. Let us begin by writing the **getDescription()** method for the **Person** class:

```
class Person {
    ...
    String getDescription() {
        return ("Person \n" +
            "   NAME:           " + this.name + "\n" +
            "   ADDRESS:          " + this.address + "\n" +
            "   PHONE #:          " + this.phoneNumber);
    }
}
```

It is quite straight forward. At this point, all of the subclasses will inherit this method and we will obtain the following results:

```
// Result when calling getDescription()
// on Person object
"Person
  NAME:           Jim Class
  ADDRESS:        1445 Porter St.
  PHONE #:        613-555-3232"

// Result when calling getDescription()
// on Secretary object
"Person
  NAME:           Earl E. Bird
  ADDRESS:        12 Knowhere Cres.
  PHONE #:        613-555-7854"

// Result when calling getDescription()
// on Employee object
"Person
  NAME:           Rob Banks
  ADDRESS:        789 ScotiaBank Road.
  PHONE #:        613-555-2332"

// Result when calling getDescription()
// on FullTimeStudent object
"Person
  NAME:           May I. Passplease
  ADDRESS:        5567 Java Drive
  PHONE #:        613-555-8923"

// Result when calling getDescription()
// on Professor object
"Person
  NAME:           Guy Smart
  ADDRESS:        267 Lost Cres.
  PHONE #:        613-555-2378"

// Result when calling getDescription()
// on PartTimeStudent object
"Person
  NAME:           Al B. Back
  ADDRESS:        23 Return Blvd.
  PHONE #:        613-555-9738"
```

So by writing the one method in **Person**, all subclasses inherit it for free. However, we would like the “type” of person displayed as the first word of the String ... we do not always want “Person” displayed. We can make use of **this.getClass().getName()** to extract the name of the particular object. So we just need to make the following change in the method that we wrote:

```
class Person {
    ...
    String getDescription() {
        return (this.getClass().getName() + "\n" +
                "  NAME:      " + this.name + "\n" +
                "  ADDRESS:   " + this.address + "\n" +
                "  PHONE #:   " + this.phoneNumber);
    }
}
```

Now all of the subclasses will inherit this method and we will obtain the following results (note the highlighted changes):

```
// Result when calling getDescription()
// on Person object
"Person
NAME:      Jim Class
ADDRESS:   1445 Porter St.
PHONE #:   613-555-3232"

// Result when calling getDescription()
// on Secretary object
"Secretary
NAME:      Earl E. Bird
ADDRESS:   12 Knowhere Cres.
PHONE #:   613-555-7854"

// Result when calling getDescription()
// on Employee object
"Employee
NAME:      Rob Banks
ADDRESS:   789 ScotiaBank Road.
PHONE #:   613-555-2332"

// Result when calling getDescription()
// on FullTimeStudent object
"FullTimeStudent
NAME:      May I. Passplease
ADDRESS:   5567 Java Drive
PHONE #:   613-555-8923"

// Result when calling getDescription()
// on Professor object
"Professor
NAME:      Guy Smart
ADDRESS:   267 Lost Cres.
PHONE #:   613-555-2378"

// Result when calling getDescription()
// on PartTimeStudent object
"PartTimeStudent
NAME:      Al B. Back
ADDRESS:   23 Return Blvd.
PHONE #:   613-555-9738"
```

This is closer to what we want. Now what else is common within the remaining classes ? We notice that all **Employees** (also **Professors** and **Secretaries**) show their employee number and work number. Therefore, this can all be done in the **Employee** class.

In order to make sure that the **name**, **address** and **phoneNumber** are shown as well, we must use the superclass method first. So we will override the **getDescription()** method in the **Person** class with one in the **Employee** class that will make use of its superclass method and then add additional code ...

```

class Employee extends Person {
    ...
    String getDescription() {
        return (super.getDescription() + "\n" +
            "  EMPLOYEE #: " + this.employeeNumber + "\n" +
            "  WORK #:      " + this.workNumber);
    }
}

```

Note the use of `super.getDescription()`. This allows us to inherit what the superclass did and then add more information for this class. Here is what we have as results so far:

```

// Result when calling getDescription()
// on Person object
"Person
  NAME:      Jim Class
  ADDRESS:   1445 Porter St.
  PHONE #:   613-555-3232"

// Result when calling getDescription()
// on Employee object
"Employee
  NAME:      Rob Banks
  ADDRESS:   789 ScotiaBank Road.
  PHONE #:   613-555-2332
  EMPLOYEE #: 88765
  WORK #:    613-555-2433"

// Result when calling getDescription()
// on Professor object
"Professor
  NAME:      Guy Smart
  ADDRESS:   267 Lost Cres.
  PHONE #:   613-555-2378
  WORK #:    613-555-3415
  OFFICE #:  5240 PA"

// Result when calling getDescription()
// on Secretary object
"Secretary
  NAME:      Earl E. Bird
  ADDRESS:   12 Knowhere Cres.
  PHONE #:   613-555-7854
  EMPLOYEE #: 76845
  WORK #:    613-555-3243"

// Result when calling getDescription()
// on FullTimeStudent object
"FullTimeStudent
  NAME:      May I. Passplease
  ADDRESS:   5567 Java Drive
  PHONE #:   613-555-8923"

// Result when calling getDescription()
// on PartTimeStudent object
"PartTimeStudent
  NAME:      Al B. Back
  ADDRESS:   23 Return Blvd.
  PHONE #:   613-555-9738"

```

Notice how the **Employee**, **Professor**, and **Secretary** classes all have the additional information now. In fact, we do not even need to write a `getDescription()` method in the **Secretary** class since it inherits the one in **Employee** which already does what it needs to do.

Now what about the **Student** class? It merely inherits from **Person** and also displays the `studentNumber`, so we can use `super` again.

```

class Student extends Person {
    ...
    String getDescription() {
        return (super.getDescription() + "\n" +
            "  STUDENT #:  " + this.studentNumber);
    }
}

```

We can append a little more for **FullTimeStudent** objects to include their major:

```
class FullTimeStudent extends Student {
    ...
    String getDescription() {
        return (super.getDescription() + "\n" +
            "    MAJOR:      " + this.major);
    }
}
```

Here is the resulting change to results when calling **getDescription()** on a **Student**:

```
// Result when calling getDescription() // Result when calling getDescription()
// on FullTimeStudent object           // on PartTimeStudent object
"FullTimeStudent                       "PartTimeStudent
NAME:      May I. Passplease           NAME:      Al B. Back
ADDRESS:   5567 Java Drive             ADDRESS:   23 Return Blvd.
PHONE #:   613-555-8923                PHONE #:   613-555-9738
STUDENT #: 100156753                   STUDENT #: 100134257"
MAJOR:     Computer Science"
```

Notice that we did not write any code in **PartTimeStudent**, as this simply inherits the method from **Student**.

The **Professor** class however, will have problems. We cannot merely inherit from **Employee** because the **Employee** method inherits from **Person** (which displays the phone number and address, and professors do not want this information available). Therefore, we can do this one without inheritance by completely overriding the **Employee** method:

```
class Professor extends Employee {
    ...
    String getDescription() {
        return ("Professor \n" +
            "    NAME:      " + this.name + "\n" +
            "    EMPLOYEE #: " + this.employeeNumber + "\n" +
            "    WORK #:    " + this.workNumer + "\n" +
            "    OFFICE# :  " + this.officeNumber);
    }
}
```



Now we have the results that we want. It is too bad though, since we now have some duplicate code in the **Professor** class. If only there was a way to make use of "some" of the code in the **Person** class, but not all of it. But if we change the **getDescription()** method in the **Person** class, then this will affect the **getDescription()** method in the **Employee** class. We must be careful.

We can create two **getDescription()** methods in the **Person** class. One that displays the **name** only, and the other one will display **name**, **address** and **phoneNumber**. Of course the methods need to have different names:

```

class Person {
    ...
    String getDescription() {
        return (this.getClass().getName() + "\n" +
            "  NAME:      " + this.name + "\n" +
            "  ADDRESS:    " + this.address + "\n" +
            "  PHONE #:    " + this.phoneNumber);
    }
    String shortGetDescription() {
        return (this.getClass().getName() + "\n" +
            "  NAME:      " + this.name);
    }
}

```

We want the **Professor** class to use the **shortGetDescription()** method ... but how do we do it? If we call **super.shortGetDescription()**, it will display the **name** properly, but then we still need to display the **employeeNumber**, **workNumber** and **officeNumber**:

```

class Professor extends Employee {
    ...
    String getDescription() {
        return (super.shortGetDescription() + "\n" +
            "  EMPLOYEE #: " + this.employeeNumber + "\n" +
            "  WORK #:    " + this.workNumber + "\n" +
            "  OFFICE# :  " + this.officeNumber);
    }
}

```

How can we make use of the **getDescription()** in the **Employee** class so that it displays the **employeeNumber** and **workNumber** for us? Well, we can make an additional **getDescription()** in the **Employee** class as follows:

```

class Employee extends Person {
    ...
    String getDescription() {
        return (super.getDescription() + "\n" +
            "  EMPLOYEE #: " + this.employeeNumber + "\n" +
            "  WORK #:    " + this.workNumber);
    }

    String shortGetDescription() {
        return (super.shortGetDescription() + "\n" +
            "  EMPLOYEE #: " + this.employeeNumber + "\n" +
            "  WORK #:    " + this.workNumber);
    }
}

```

But this duplicates code in the **Employee** class!!! We can fix this by extracting the common code into a helper method as follows:

```
class Employee extends Person {
    ...
    String commonString() {
        return ("\n" + " EMPLOYEE #: " + this.employeeNumber +
            "\n" + " WORK #: " + this.workNumber);
    }

    String getDescription() {
        return(super.getDescription() + this.commonString());
    }

    String shortGetDescription() {
        return(super.shortGetDescription() + this.commonString());
    }
}
```

Now the **Professor** class merely needs to call the **shortGetDescription()** method as before ... but this time the **super** call finds the method in the **Employee** class, so that one is executed (which in turn calls the one in **Person**). Here is the code:

```
class Professor extends Employee {
    ...
    String getDescription() {
        return(super.shortGetDescription() + "\n" +
            " OFFICE# : " + this.officeNumber);
    }
}
```

This is nice and short now and makes use of shared code. In fact, we do not even need to say **super** in the code above, and it will work the same. Do you know why?

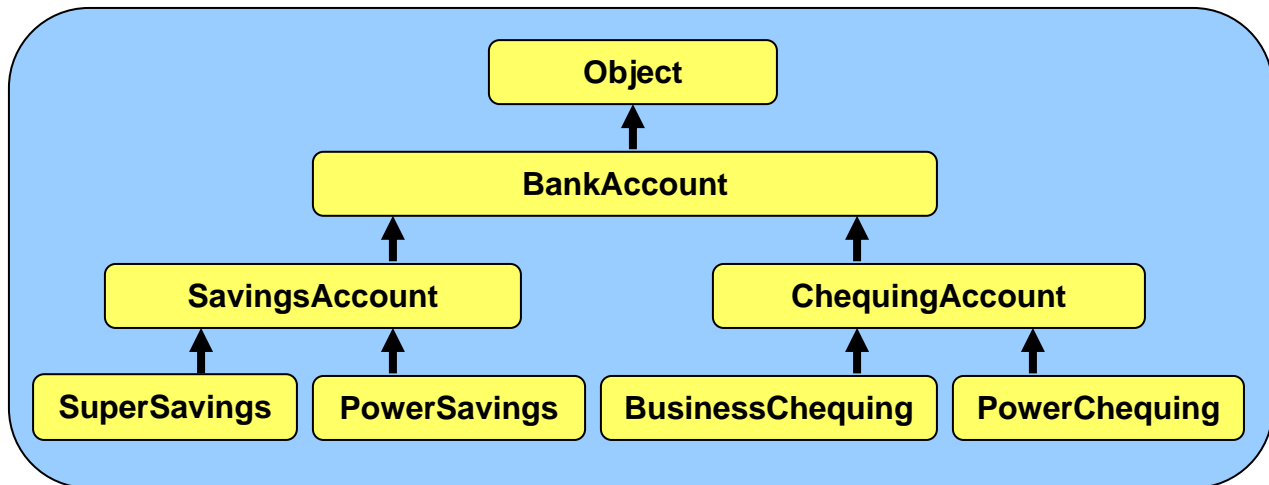
## 7.5 Abstract Classes & Methods

Recall our example in the previous section pertaining to the various types of bank accounts. We had two types of accounts: **SuperSavings** and **PowerSavings**, which both inherited from a more general class called **SavingsAccount** and indirectly from **BankAccount** a little further up the hierarchy. Assume further that we distinguished between savings accounts and chequing accounts ... where chequing accounts allow their owners to write cheques.

Assume that the real bank actually has exactly 4 types of accounts so that when someone goes to the bank teller to open a new account, they specify whether or not they want to open a **SuperSavings**, **PowerSavings**, **BusinessChequing** or **PowerChequing** account.



Here is a revised hierarchy ...



In our class hierarchy however, there are 7 account-related classes. The four classes representing the accounts that we can actually open are called **concrete** classes. A **concrete** class in JAVA is a class that we can make instances of directly by using the **new** keyword. That is, throughout our code, we will find ourselves creating one of these 4 classes. For example:

```

account1 = new SuperSavings(...);
account2 = new PowerSavings(...);
account3 = new BusinessChequing(...);
account4 = new PowerChequing(...);
  
```



However, we will likely never need to create instances of the other 3 account-related classes:

```

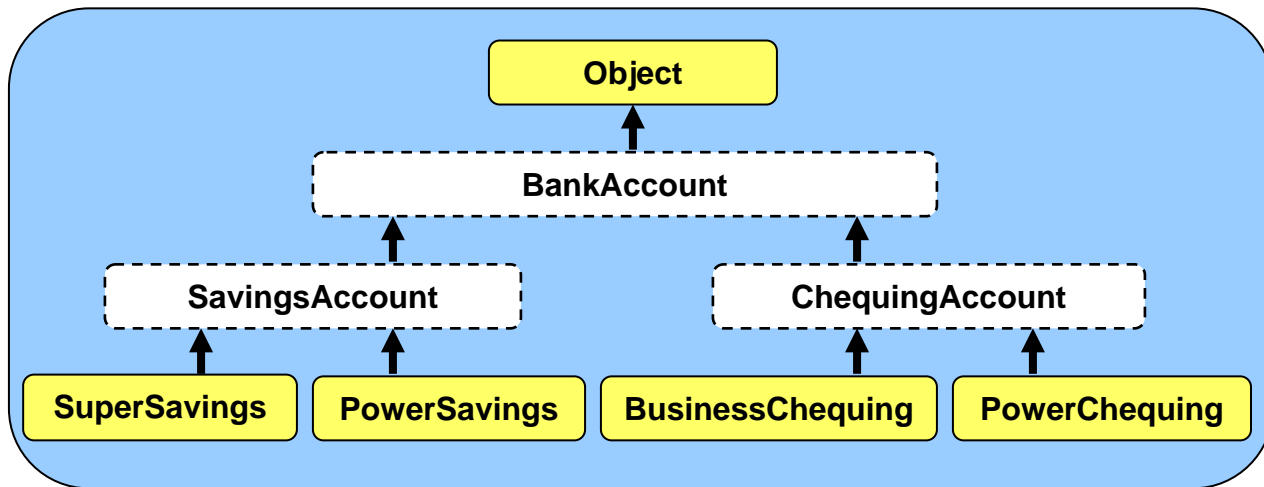
account5 = new BankAccount(...);
account6 = new SavingsAccount(...);
account7 = new ChequingAccount(...);
  
```



Why not? Well, put simply, these types of objects are not specific enough because they cause ambiguity. For example, if you went to the bank teller and asked to open just “a bank account”, the teller does not know which of the 4 types of accounts you actually want. The teller would likely ask you questions to help you narrow down your choices, but ultimately, the type of account that is opened (i.e., the account that is actually created) **MUST** be one of the 4 accounts that the bank offers. Likewise, in our program, if we were to create instances of **BankAccount**, **SavingsAccount** and **ChequingAccount**, then these objects would not be specific enough to define account behavior that matches one of the 4 real account types.

So in a sense, the **BankAccount**, **SavingsAccount** and **ChequingAccount** classes are not concrete, they are more abstract in that they don't exactly match the real-life objects. In JAVA, we actually use the term **abstract class** to define a class that we do not want to make instances of. So, **BankAccount**, **SavingsAccount** and **ChequingAccount** should all be abstract classes. We will draw abstract classes with dotted lines as follows ...





So, in JAVA, an **abstract** class is simply a class for which we cannot create instances. That means, we can never call the constructor to make a new object of this type.

```

new BankAccount(...) // does not compile
new SavingsAccount(...) // does not compile
new ChequingAccount(...) // does not compile
  
```



All of the classes that we created so far in this course were concrete classes, although some could have been easily made abstract. We define a class to be abstract simply by using the **abstract** keyword in the class definition:

```

abstract class BankAccount {
    ...
}
  
```

```

abstract class SavingsAccount extends BankAccount {
    ...
}
  
```

```

abstract class ChequingAccount extends BankAccount {
    ...
}
  
```

That is all that is involved in creating an abstract class. There really is nothing more to it. In fact, the remainder of the code in that class definition may remain as is.

So, in fact, by making a class abstract, all we have done is to prevent the user of the class from calling any of its constructors directly. This may raise an interesting question. If we cannot ever create new objects of the abstract class, then why would we ever want to create an **Abstract** class in the first place ?

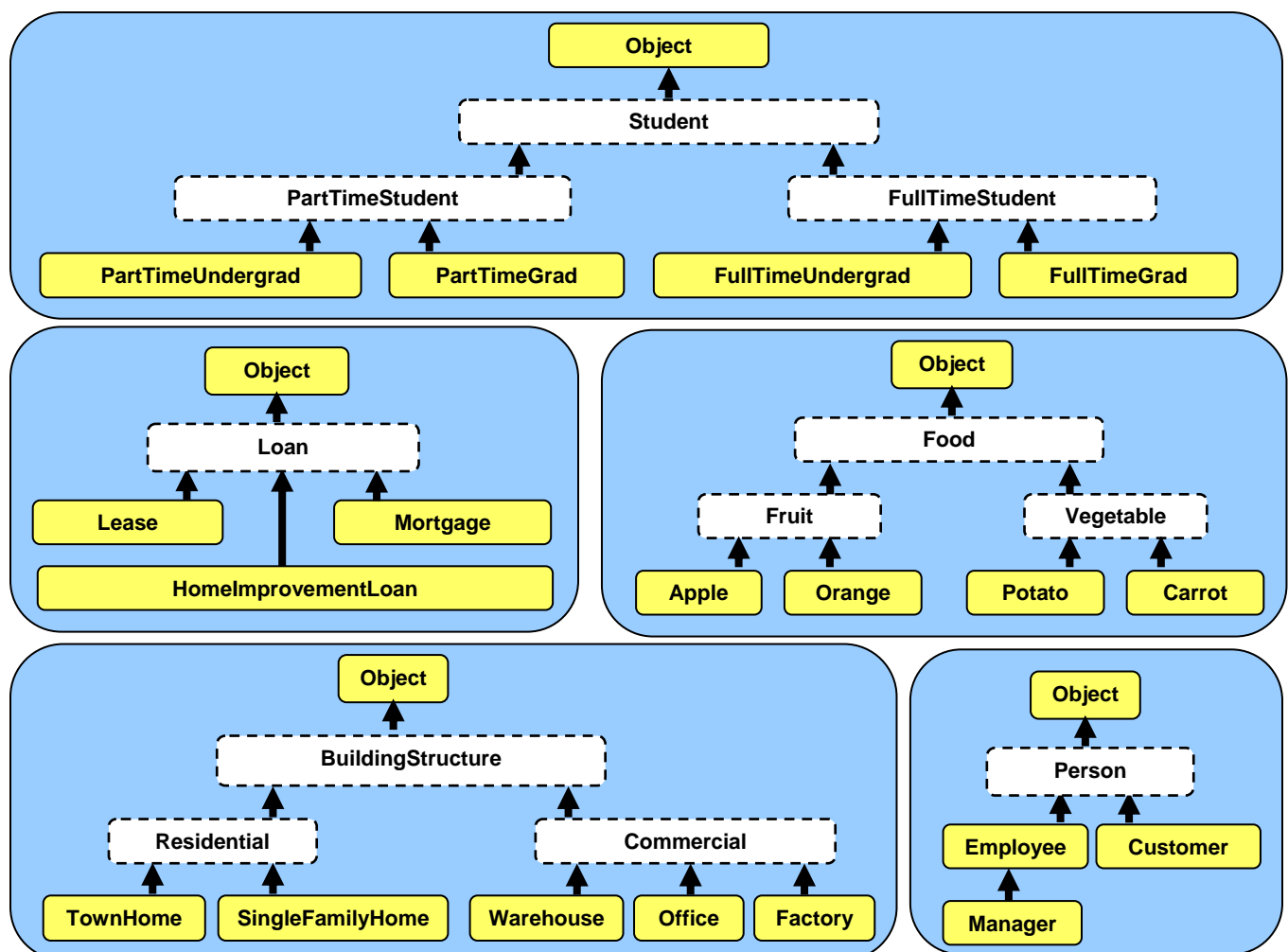
Well why did we create the **BankAccount** and **SavingsAccount** classes in the first place ? Inheritance was the key reason. These classes still contain the common attributes and shared behavior for all of their subclasses. The **BankAccount** class, for example, contains the 3 instance variables common to all accounts (**owner**, **accountNumber** and **balance**).

Also, the **SavingsAccount**, for example, contains the **deposit()** method that is shared between **SuperSavings** and **PowerSavings**. Hence, you can see that even though a class may be declared as **abstract** it is still useful and important in keeping our code organized properly in our class hierarchy. Their attributes and behaviors are still being used by their concrete subclasses.

How do we know which classes to make *abstract* and which ones to leave as *concrete*? If we are not sure, it is better to leave them as concrete. However, if we discern that a particular class has subclasses that cover all of the possible concrete classes that we would ever need to create in our application, then it would be reasonable to make the superclass abstract.

Is there any advantage of making a class *abstract* rather than simply leaving it *concrete*? Yes. By making a class *abstract*, you are informing the users of that class that they should not be creating instances of that class. In a way, you are telling them “**If you want to use this class, you should make your own concrete subclass of it.**”. You are actually *forcing* them to create a subclass if they want to use your abstract class. It forces the user of your class to be more specific in their object creation, thereby **reducing ambiguity** in their code.

Here are a few more examples of class hierarchies that we already discussed, showing how we could make some classes abstract:



## Abstract Methods:

In addition to having **abstract** classes, JAVA allows us to make **abstract methods**. An abstract method is a method which has no code. That is, it is merely a specification of a method's signature (i.e., return type, name and list of parameters), but the body of the code remains blank. To define an abstract method, we use the **abstract** keyword at the beginning of the method's signature. Here are a couple of examples:

```
abstract void deposit(float amount);  
abstract void withdraw(float amount);
```

Notice that there are no braces **{ }** to specify the method body ... the method signature simply ends with a semi-colon **;**.

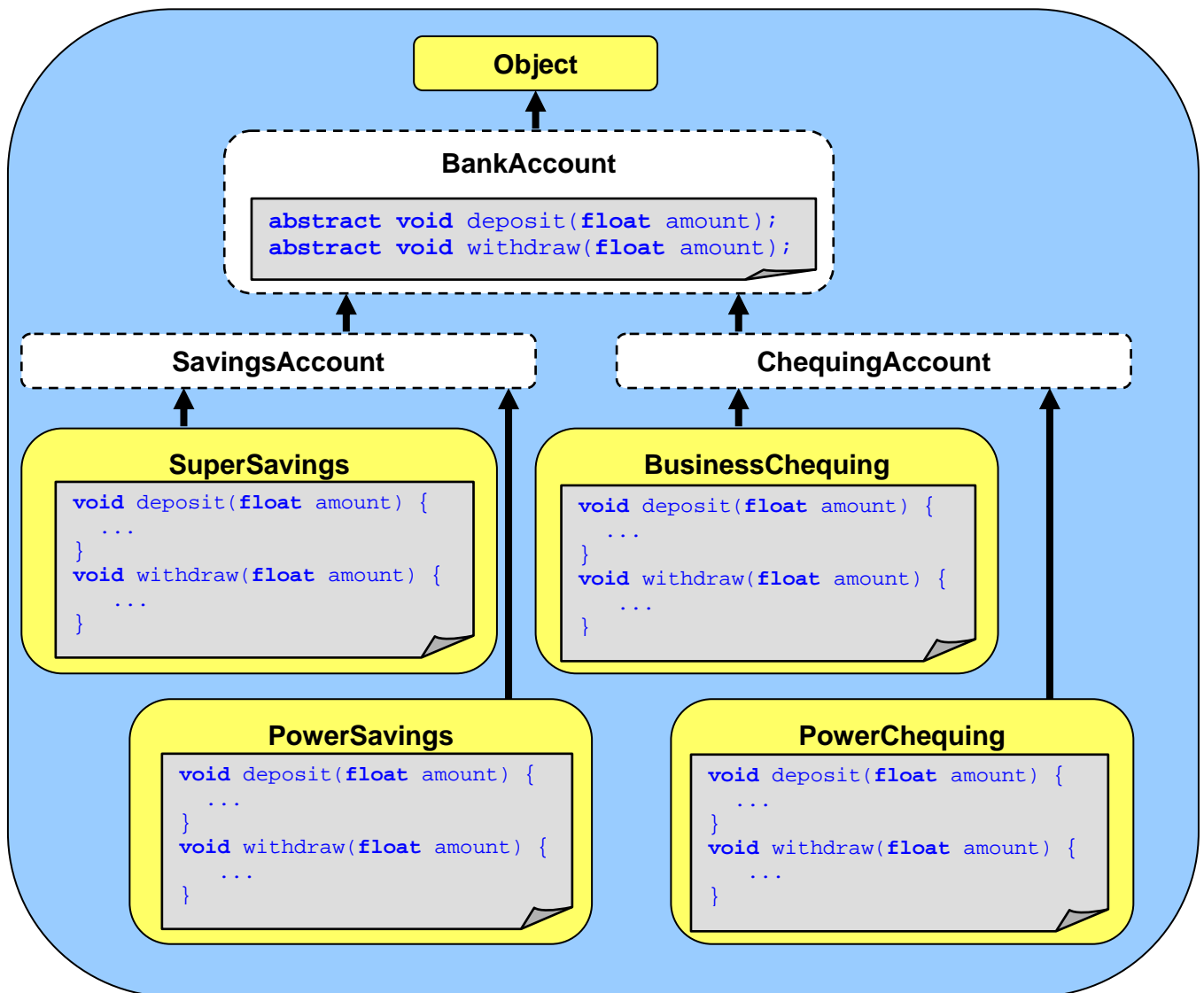
At this point you should be wondering: “**Why would any sane person would write a method that has no code in it ?**”. That is certainly a reasonable question since, after all, methods are called so that we can evaluate the code that is in them.

Abstract methods are actually never called, so JAVA never attempts to evaluate their code. Just as an abstract class is used to force the user of that class to have subclasses, an abstract method *forces* the subclasses to **implement** (i.e., to write code for) that method. So, by defining an abstract method, you are really just informing everyone that the concrete subclasses must write code for that method. All concrete subclasses of an **abstract** class **MUST** implement the **abstract** methods defined in their superclasses, there is no way around it.

When JAVA compiles an abstract method for a class (e.g., class **A**), it checks to see whether or not all the subclasses of **A** have implemented the method (i.e., that they have written a method with the same return type, name and parameters). That is really all that happens in regard to the abstract methods.

For example, if we make **deposit(float amount)** and **withdraw(float amount)** methods **abstract** in the **BankAccount** class, then, all of its concrete subclasses (**SuperSavings**, **PowerSavings**, **BusinessChequing** and **PowerChequing**) would be forced to implement those methods ... complete with code as follows ...

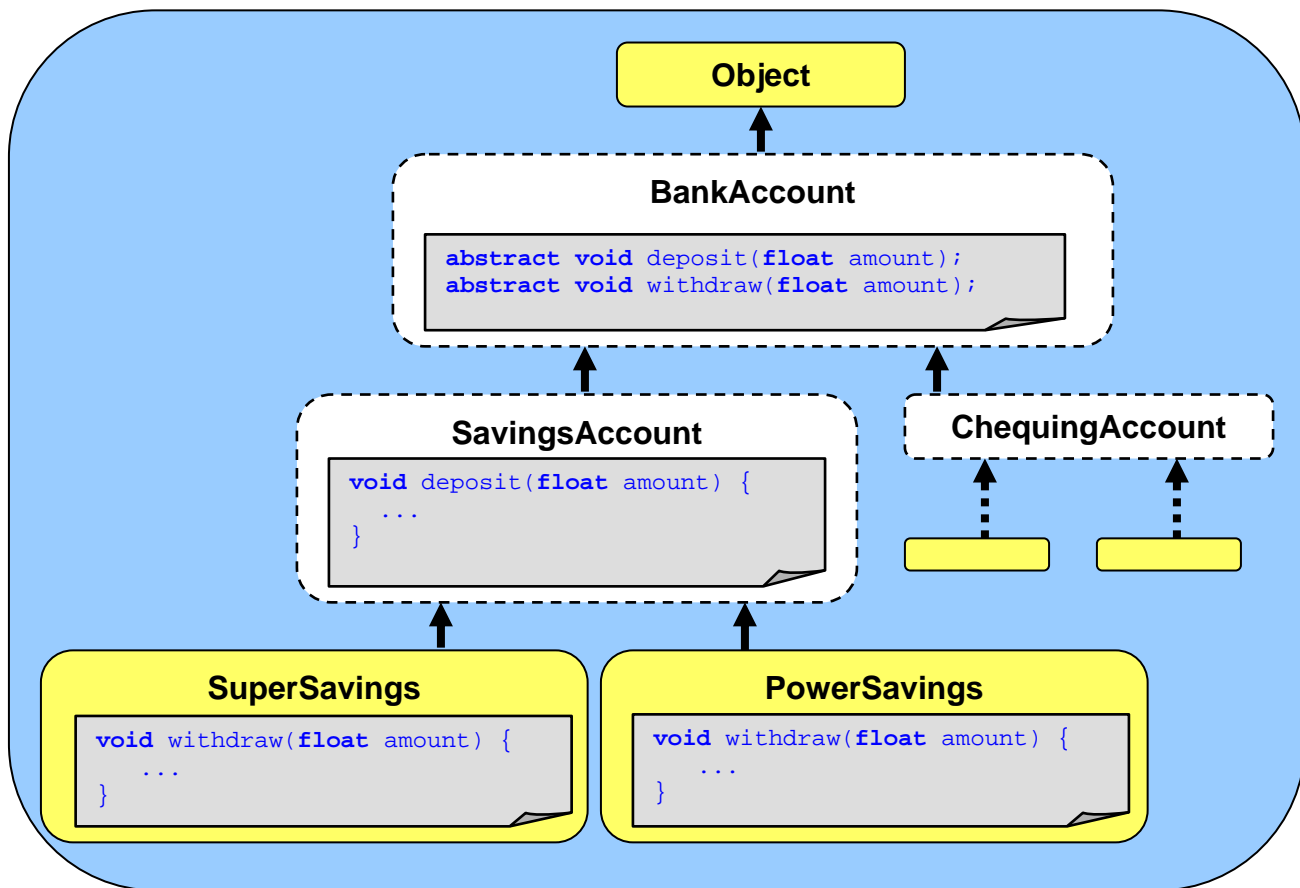




Each of the 4 concrete subclasses would implement their **deposit()** and **withdraw()** code according to the bank's rules for that type of account (i.e., apply certain fees, limit amount, etc...).

Alternatively, we can take advantage of inheritance. If, for example, the **SuperSavings** and **PowerSavings** accounts both **deposit()** in the same manner, instead of duplicating the code we can implement a non-abstract **deposit()** method in the **SavingsAccount** class that performs the required behavior. This method would then be shared (i.e., used) by both the **SuperSavings** and **PowerSavings** subclasses through inheritance.

In this case, the **SuperSavings** and **PowerSavings** classes would NOT need to implement the **deposit()** method, since it is inherited ...

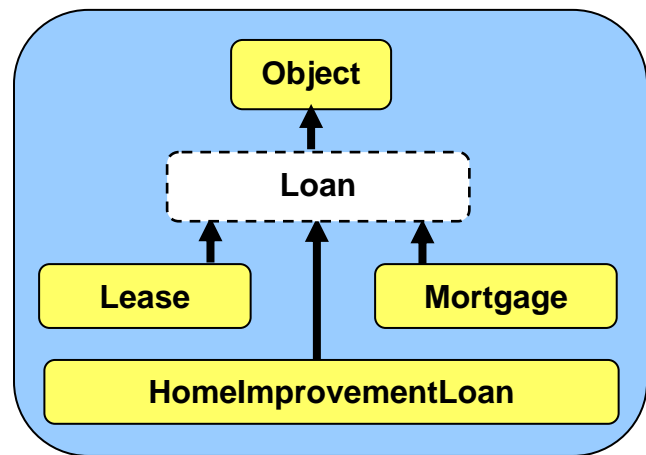


Only **abstract** classes are allowed to have such abstract methods. However, as you know, an **abstract** class may have regular methods as well.

If we were to find that all 4 types of concrete accounts did the exact same thing when a **deposit()** was made, then we would likely simply write the shared **deposit()** method in the **BankAccount** class, INSTEAD OF making the abstract **deposit()** method in the first place. This allows a kind of default **deposit()** behavior for all subclasses to inherit, not forcing any classes to implement this method.

It is often the case that we define more than one abstract method in a class. This allows us to **specify** a set of “standard” behavior that ALL of its subclasses MUST have. For example, assume that we have the following hierarchy in which an abstract **Loan** class has 3 specific subclasses as shown here →

We may decide on some particular behavior that all types of loans must exhibit. For example, we may want to ensure that we have a way to calculate a monthly payment for the loan, a way to make payments on the loan, a way to re-finance the loan and perhaps a way to extract the client’s information that pertains to the loan.



If this is the case, perhaps some of the behavior is similar for all loans (e.g., getting the client's information), while other behaviors may be unique depending on the type of loan (e.g., leases and mortgages may be re-financed differently). Here is how we might define the **Loan** class:

```
abstract class Loan {
    abstract float calculateMonthlyPayment();
    abstract void makePayment(float amount);
    abstract void renew(int numMonths);

    Client getClientInfo() { // a non-abstract method
        ...
    }
    ....
}
```

Notice that the **getClientInfo()** method is non-abstract, so that we can write code in there that is shared by all the subclasses. The other 3 methods shown are **abstract** ... so the **Lease**, **Mortgage** and **HomeImprovementLoan** classes **MUST** implement all 3 of these methods, with the appropriate code. Remember ... an abstract class is just like any other class in regards to its attributes and behaviors. So there may be many more methods (abstract or non-abstract) and/or attributes defined in the **Loan** class.

Do you see the benefit of defining abstract methods? They allow you to define a set of behaviors that all your subclasses **must have** while giving them the flexibility to specify their **own unique code** for those behaviors. What would happen if we did not make any of the methods abstract?:

```
abstract class Loan {
    float calculateMonthlyPayment(){ return 0;}
    void makePayment(float amount){ }
    void renew(int numMonths){ }
    Client getClientInfo() { ... }
    ....
}
```

Two things would be different. First, the methods would need to have a body. We could leave the code body blank or we could put in some default code of our choosing.

Second, the subclasses would not be "*forced*" to write these methods. So if the subclass did not supply the method, then these methods here would be inherited. This is not such a "big deal", but if we simply *forgot* to implement these methods, then the inherited behavior may be unexpected and in some cases undesirable. By making the 3 methods **abstract**, the compiler will *force* us to write the methods, eliminating the possibility of us forgetting to implement them.

## 7.6 Defining Interfaces

In the previous section, we showed how we can use a set of **abstract methods** to force a set of specific behaviors in our subclasses. By doing that, we were able to ensure that all of the subclasses of that **abstract** class have definitely implemented the required behavior.

Abstract methods are indeed quite useful for defining (and forcing) common behavior for subclasses of an **abstract** class. How though, would we define (and force) common behavior between seemingly *unrelated* classes in different parts of the class hierarchy ?

There is another mechanism in JAVA for doing this. In JAVA, an **interface** is a specification (i.e., a list) of a set of behaviors. It is similar to the idea of having a set of abstract methods, except that the interface exists on its own, that is, it is defined by itself in its own file. We define this list of behaviors/methods as if we were defining a new class, except that we use the keyword **interface** instead of **class**:

```
interface InterfaceName {
    ...
}
```

Just like classes, interfaces are *types* and are defined in their own .java files. So, the above interface would be saved into a file called **InterfaceName.java**.

The methods themselves are defined like abstract methods, but without the word abstract. Here is a comparison of the **Loan** class's abstract methods and a similarly-defined interface:

```
abstract class Loan {
    abstract float calculateMonthlyPayment();
    abstract void makePayment(float amount);
    abstract void renew(int numMonths);
    ...
}

interface Loanable {
    float calculateMonthlyPayment();
    void makePayment(float amount);
    void renew(int numMonths);
}
```

There are some **similarities** between the two:

- both define a list of methods with no code.
- like abstract classes, we cannot create instances of interfaces. So, we cannot use this code anywhere in our code: `new Loan()`

There are also some **differences** between the two:

- we cannot declare/define any attributes nor static constants in an interface, whereas an abstract class may have them
- we can only declare “empty” methods in an interface, we cannot supply code for them. In contrast, an abstract class can have non-abstract methods with complete code.

Since interfaces are defined on their own (i.e., the interface does not *belong* to any particular class), we must have a way to inform JAVA which objects will be implementing the methods that are defined in the interface.

Consider defining an interface called **Insurable** that defined the common behavior that all insurable objects **MUST** have as follows:

```
interface Insurable {
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium();
    public java.util.Date getExpiryDate();
}
```

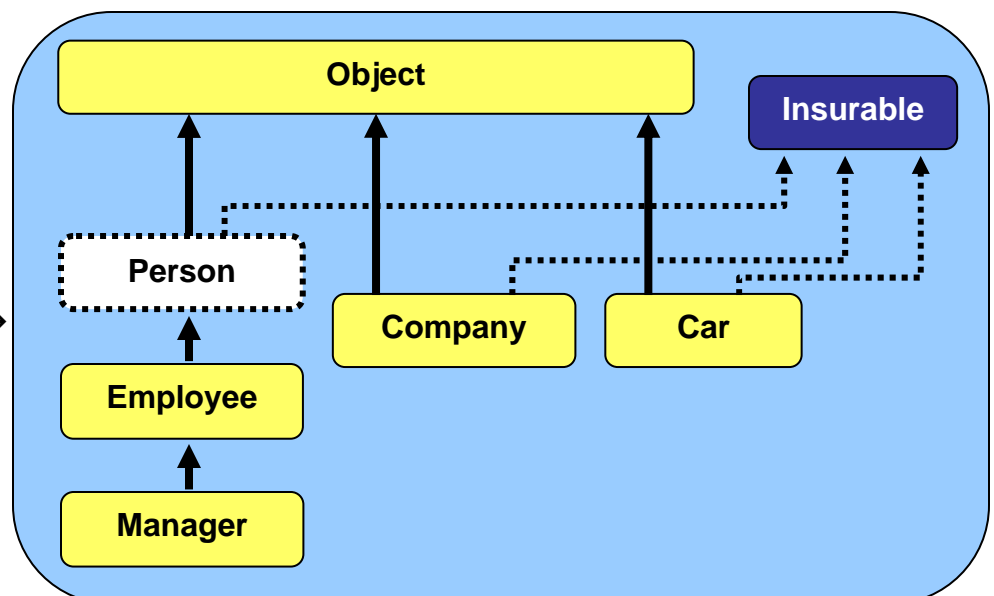


In this example, we did not have any parameters for the methods, but you can have parameters as well, just like any other method. The code above would need to be saved and compiled before we can use it. Notice that the methods all have **public** in front of them. Interfaces must be publicly accessible. We will talk more about this in an upcoming chapter.

Assume now that we want to have some classes in our hierarchy that are considered to be insurable. Perhaps **Person**, **Car** and **Company** objects in our application are all considered to be insurable objects.



We would want to make sure that they all implement the methods defined in the **Insurable** interface as shown here →





To do this in JAVA, we simply add the keyword **implements** in the class definition, followed by the **name** of the interface that the class will implement as follows:

```
abstract class Person implements Insurable {
    ...
}
```

```
class Company implements Insurable {
    ...
}
```

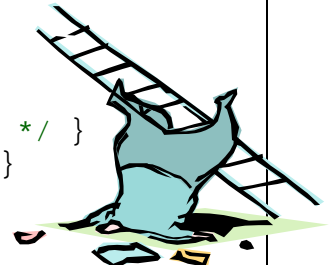
```
class Car implements Insurable {
    ...
}
```

By adding this to the top of the class definition, we are informing the whole world that these objects are insurable objects. It represents a "stamp of approval" to everyone that these objects are able to be insured. It provides a "guarantee" that these classes will have all the methods required for insurable items (i.e., **getPolicyNumber()**, **getCoverageAmount()**, **calculatePremium()** and **getExpiryDate()**). So then, for each of the implementing classes, we must go and write the code for those methods:

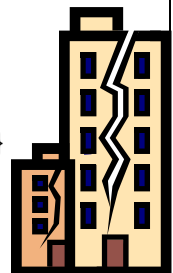
```
class Car implements Insurable {
    ...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium() { /* write code here */ }
    public java.util.Date getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /* write code here */ }
    ...
}
```



```
abstract class Person implements Insurable {
    ...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium() { /* write code here */ }
    public java.util.Date getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /* write code here */ }
    ...
}
```

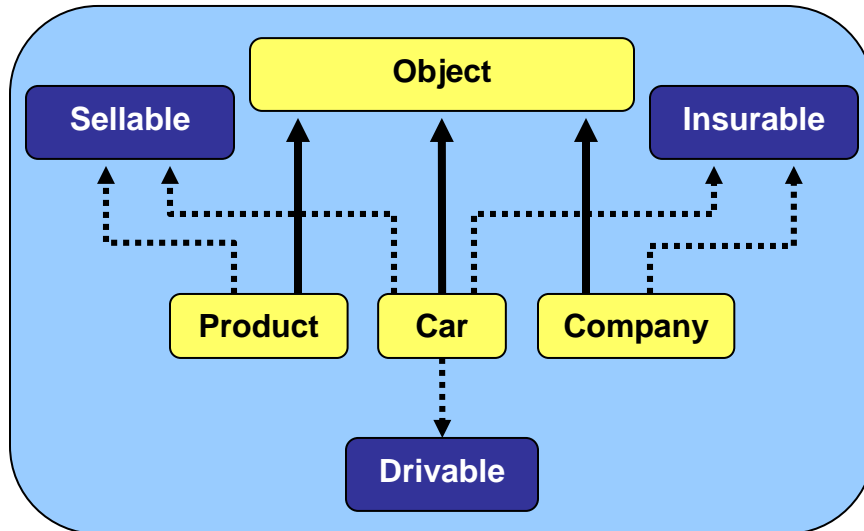


```
class Company implements Insurable {
    ...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium() { /* write code here */ }
    public java.util.Date getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /* write code here */ }
    ...
}
```



Again, notice that the methods all have **public** in front of them. We will talk more about this in an upcoming chapter. Remember that these classes may define their own attributes and methods but somewhere in their class definition they must have ALL 4 methods listed in the **Insurable** interface.

Interestingly, a class may implement more than one interface:

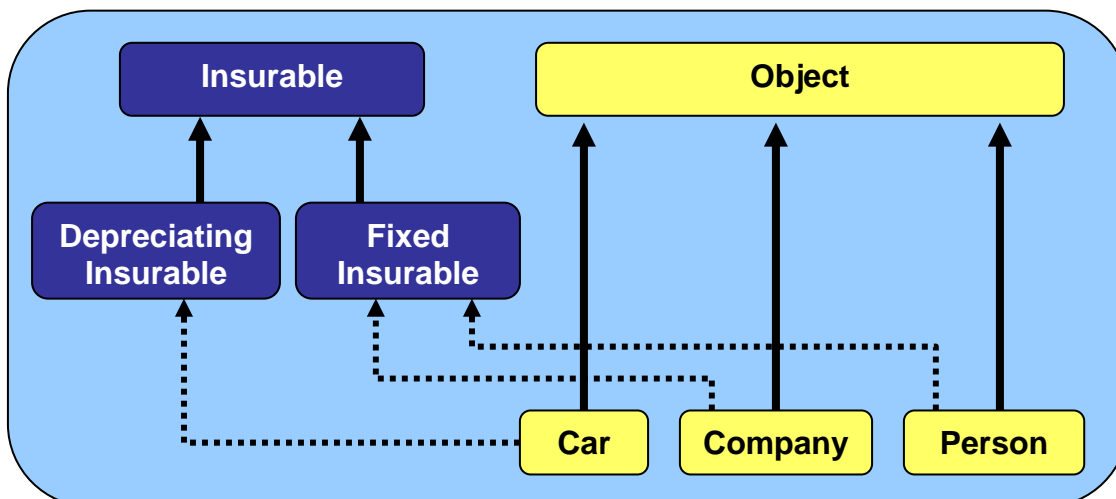


Here the **Car** object implements 3 interfaces. To allow this in our code, we just need to specify each implemented interface in our class definition (in any order), separated by commas:

```

class Car implements Insurable, Drivable, Sellable {
    ...
}
  
```

Of course, the **Car** class would have to implement **ALL** of the methods defined in **each** of the three interfaces. Like classes, interfaces can also be organized in a hierarchy:



As with classes, we form the interface hierarchy by using the **extends** keyword:

```
interface Insurable { ...
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium();
    public java.util.Date getExpiryDate();
}
```

```
interface DepreciatingInsurable extends Insurable {
    public double computeFairMarketValue();
    public void amortizePayments();
}
```

```
interface FixedInsurable extends Insurable {
    public int getEvaluationPeriod();
}
```

Classes that implement an interface must implement its "super" interfaces as well. So **Company** and **Person** would need to implement the method in **FixedInsurable** as well as the four in **Insurable**, while **Car** would have to implement the two methods in **DepreciatingInsurable** and the four in **Insurable** as well.

In summary, how do interfaces help us ? They provide us with a way in which we can specify common behavior between arbitrary objects so that we can ensure that those objects have specific methods defined. There are many pre-defined interfaces in JAVA and you will see them used often in the next course COMP1006/1406.