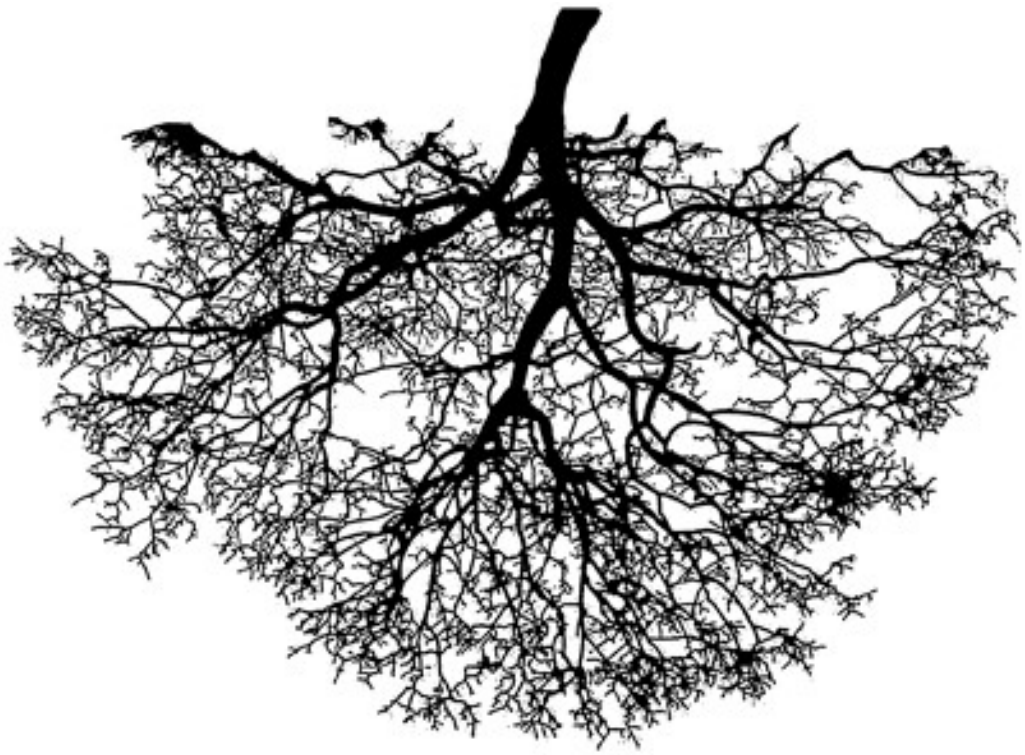
Chapter 4

Class Hierarchies and Inheritance

What is in This Chapter ?

This chapter discusses how objects are organized into a **class hierarchy** and then explains the notion of **inheritance** as a means of sharing attributes and behaviors among classes. It also explains the notion of **abstract classes** and java **interfaces** that allow seemingly unrelated classes to share common behavior.



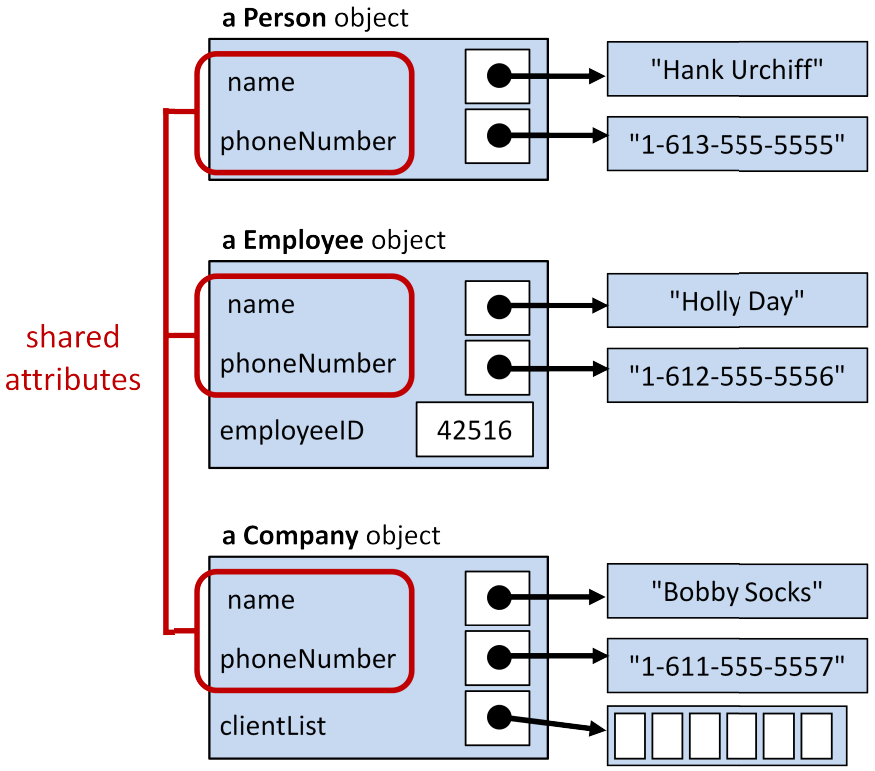
4.1 Organizing Classes

As we have already seen, defining objects as a new kind of data structure simply involves creating new ***classes***, each in their own file (e.g., Car, Person, Address, Bank, etc..). In fact, a definition of the word '**class**' in English is:

> "*A collection of things sharing a common attribute*".

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

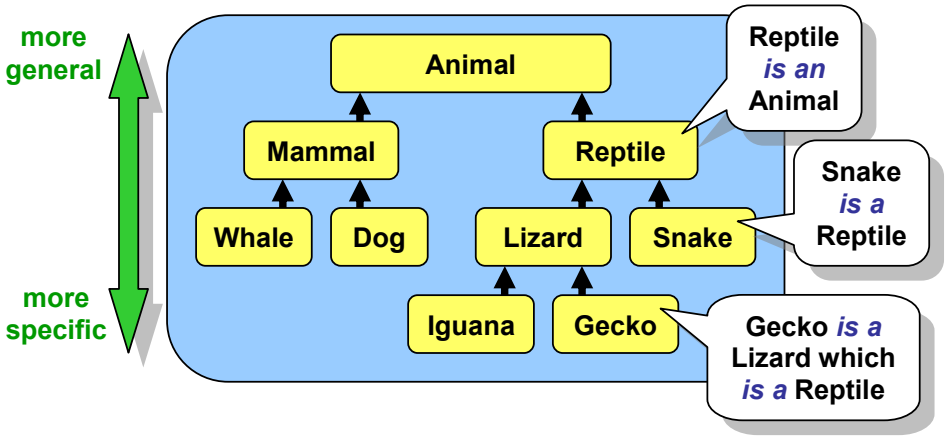
In real life, however, there are some objects that "share" attributes in common. For example, **Person** objects may have **name** and **phoneNumber** attributes, but so can **Employee**, **Manager**, **Customer** and **Company** objects. Yet, there may be additional attributes of these other objects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



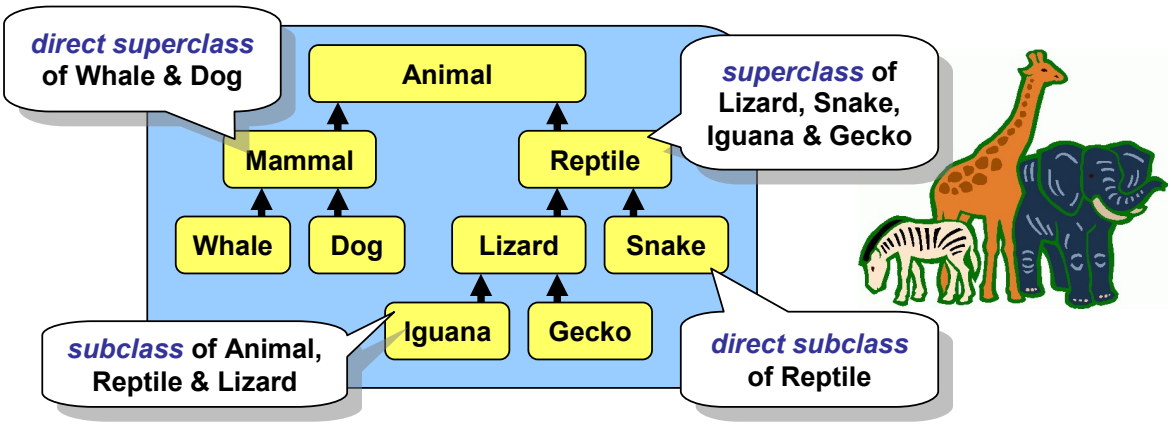
In addition to commonality between attributes, classes may also share common *behavior*. That is, two or more objects may have the ability to perform the same function or procedure. For example, if a **Person**, **Car** and **Company** are all *insurable*, then they may all have a function called **calculateInsurancePremium()** 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:



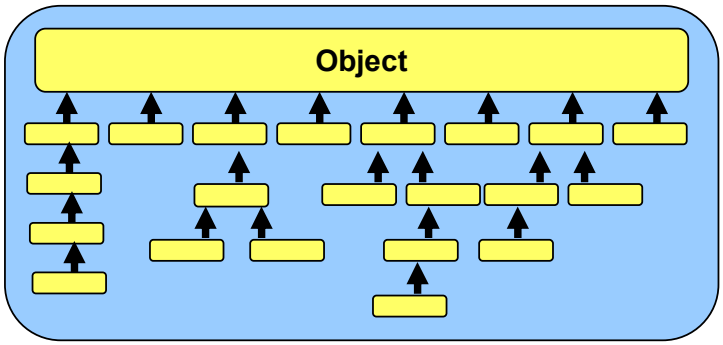
Each class is a ***subclass*** (i.e., a specialization) of some other class which is called its ***superclass*** (i.e., a generalization). The ***direct superclass*** is the class right "above" it:



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).

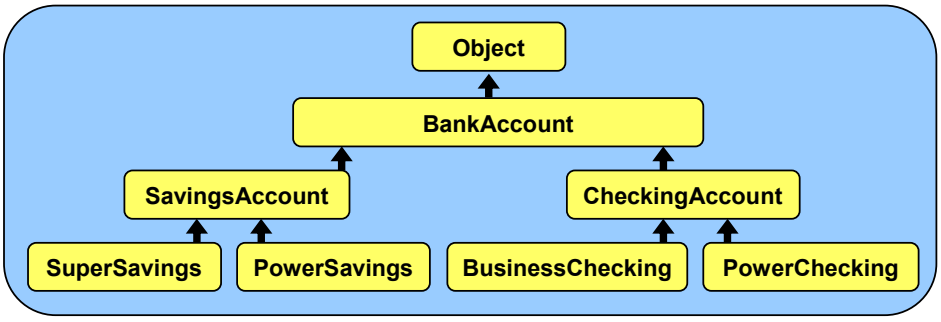
The **Animal** hierarchy above represents a set of classes that we may define ourselves. But where do they fit-in with all the other pre-made JAVA classes like **String**, **Date**, **Rectangle** etc... ? Well, all objects have one thing in common ... they are all *Objects*. Hence, at the very top of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



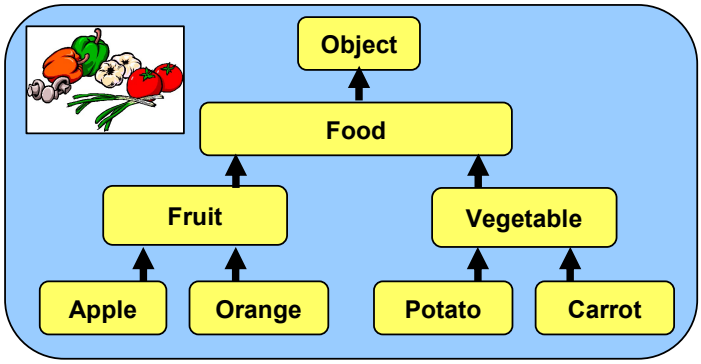
All of the classes that we created so far have been ***direct*** subclasses of **Object**. That means that they did not share attributes with one another, but that they shared attributes only with **Object**. However, we have the freedom to re-arrange our classes in a manner that will allow them to share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

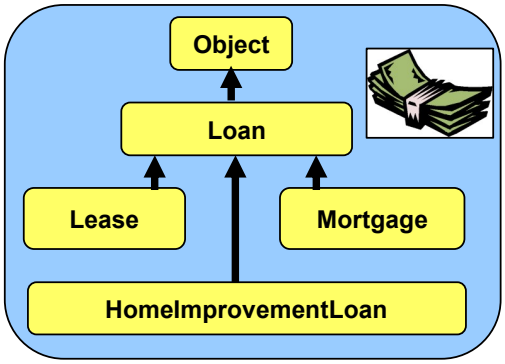
The way in which we arrange our classes will depend on *how similar* our objects are with respect to their attributes. For example, **a Car** and a **Truck** have something in common ... they are both *drivable*. Whereas an **MP3Player** and a **BankAccount** have little or nothing in common with **Car**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

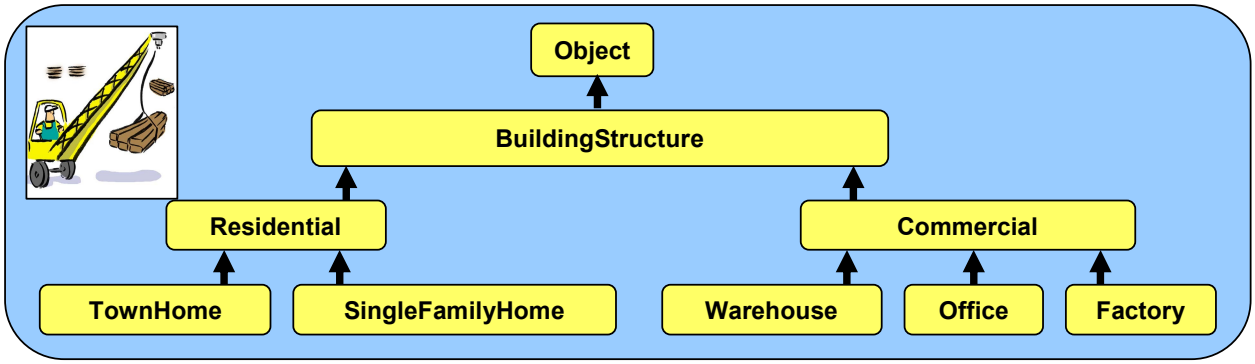
As an example, consider creating many kinds of bank accounts. We might arrange them in a hierarchy like this:



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



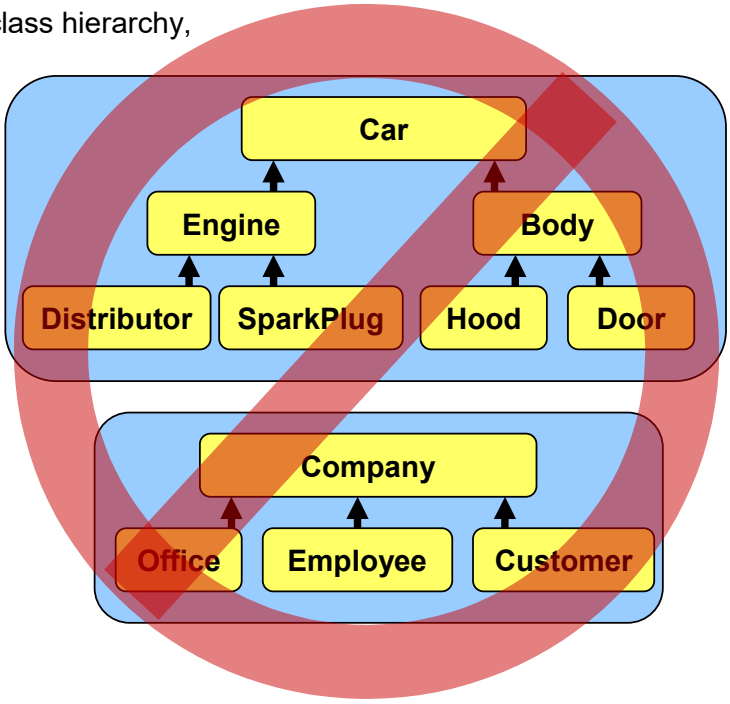




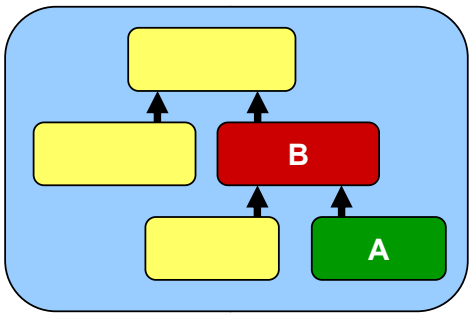
We will talk more about **how** and **why** we arrange these classes as above. But remember, a class should only be a subclass of another class if it "***is a kind of***" its superclass.

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 shown here.



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

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

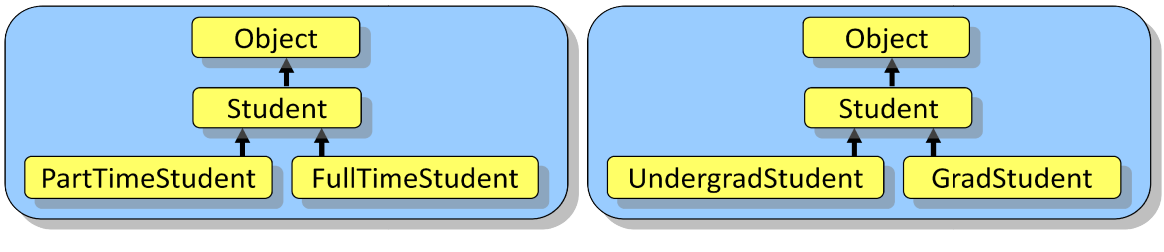
If the **extends** keyword is not used (i.e., as we left it out from all our previous class definitions), it is assumed that the class being defined extends the **Object** class. So, all the classes that we defined previously were direct subclasses of **Object**.

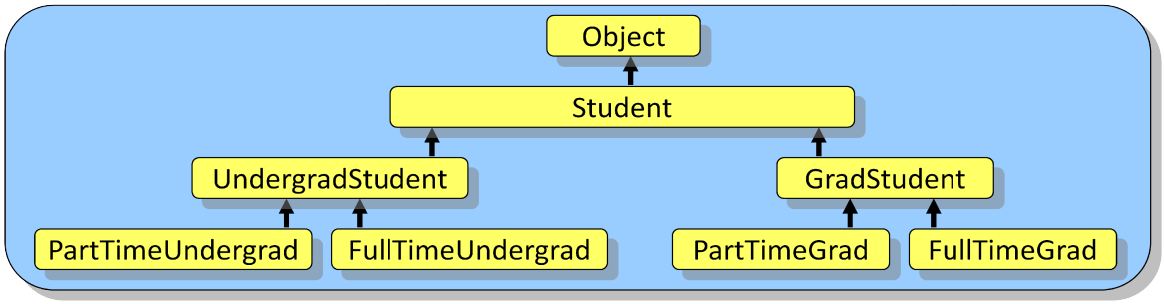
How do we know how deep we should make the class hierarchy (i.e., tree) ?

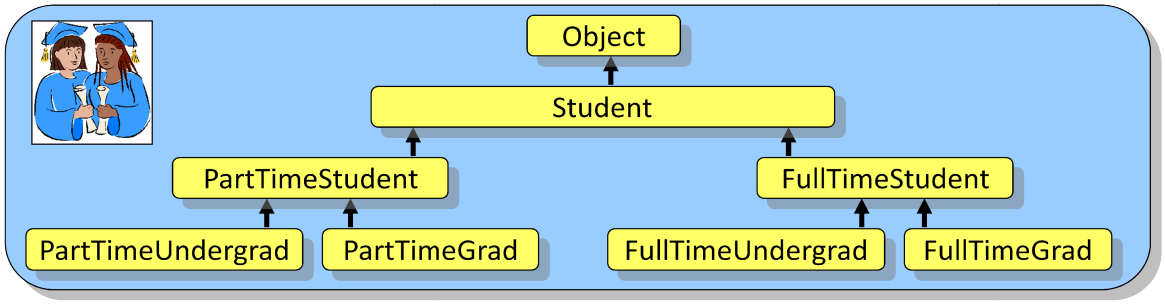
Most of the time, any **"is a"** relationship between objects should certainly result in the creation of a subclass. Object-oriented code usually involves a lot of small classes as opposed to a few large ones.

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 4 possibilities …





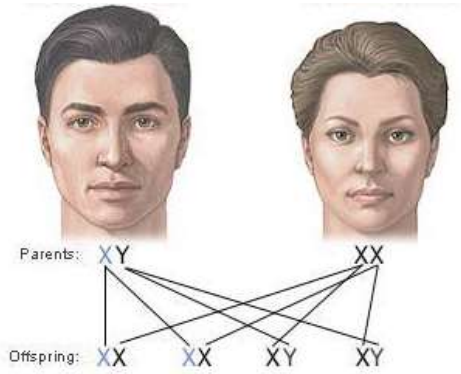


How do we know which one to use ? It will depend on the state (i.e., attributes) and behavior (i.e., methods) that is common between the subclasses. If we find that the main differences in attributes or behavior are between full time and part time students (e.g., fee payment rules), then we may choose 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.

4.2 Inheritance

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
  ... 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:

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

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:

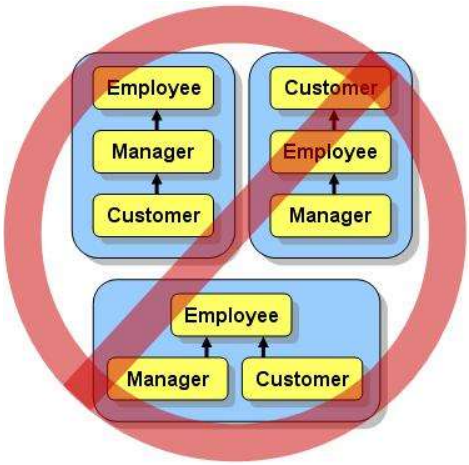
```
public class Employee {
      String      name;
      Address     address;
      String      phoneNumber;
      int         employeeNumber;
      float       hourlyPay;
      …
}

public class Manager extends Employee {
      String[]    duties;
      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:

```
public class Customer {
      String      name;
      Address     address;
      String      phoneNumber;
      …
}
```

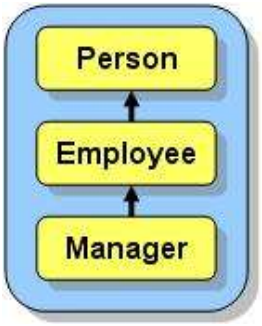
This would work fine. However, you will notice that both **Employee** and **Customer** have some attributes in common. So, if we defined the **Customer** class in this manner, we would need to repeat the same definitions, and perhaps some of the behaviors. It would be better if we could somehow use inheritance 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 (1) a **Customer** is *not always* a **Manager**, (2) an **Employee** is *not always* a **Customer**, and (3) 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:

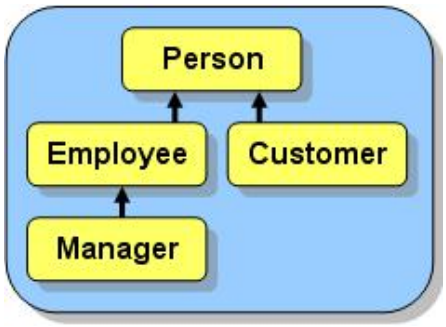
```
public class Person {
      String      name;
      Address     address;
      String      phoneNumber;
}
public class Employee extends Person {
      int         employeeNumber;
      float       hourlyPay;
}
public class Manager extends Employee {
      String[]      duties;
      Employee[]    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, phoneNumber) 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* **Person** class to hold the common attributes. We can create the following hierarchy:

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

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

public class Customer extends Person {
      String[]    itemsPurchased;
      Date[]      purchaseHistory;
}

public class Manager extends Employee {
      String[]     duties;
      Employee[]   subordinates;
}
```



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[0] = "Pencil Case";  // own attribute
c.purchaseHistory[0] = 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[0] = "Phone Clients";        // own attribute
m.subordinates[0] = 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.

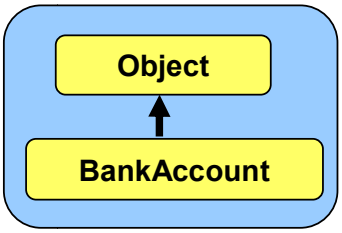
At this point, we only examined how to decide upon a class hierarchy based on the differences in attributes. However, we would have to think in the same manner by examining the behaviors of the individual classes. For example, even if managers did not have the duties and subordinates attributes shown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

Now, we will consider an example that shows how inheritance applies to behaviors within a simple hierarchy of **BankAccount** objects.

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 accountNumber 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:

```
public class BankAccount {
      String     owner;          // person who owns the account
      int        accountNumber;  // the account number
      float      balance;        // amount of money currently in the account

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

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

      // Withdraw money from the account
      public 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**:

```
public 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 **deposit()** and **withdraw()** methods.

We could verify this by writing a simple piece of test code:

```
SavingsAccount   s = new SavingsAccount();

System.out.println(s.balance);         // displays 0.0

s.deposit(120);
System.out.println(s.balance);         // displays 120.0

s.withdraw(20);
System.out.println(s.balance);         // displays 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. Then, how 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:

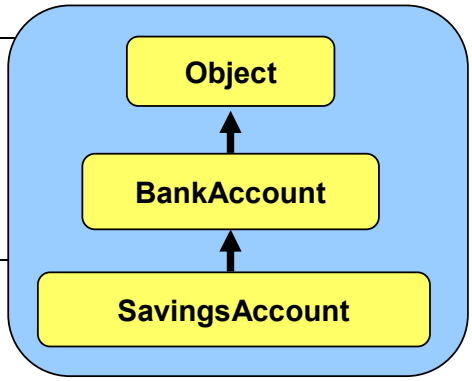
```
public SavingsAccount() {
      super();
}
```

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:

```
public class SavingsAccount extends BankAccount {
      public 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 code)	**With Inheritance**

`public SavingsAccount() { {`
`  this.owner = "";`
`  this.accountNumber = 0;`
`  this.balance = 0;`
`}`
 	
`public SavingsAccount(String ownerName) {`
`  this.owner = ownerName;`
`  this.accountNumber = 0;`
`  this.balance = 0;`
`}`
	`public SavingsAccount() {`
	`  super("");`
	`}`
	
	`public SavingsAccount(String aName) {`
	`  super(aName);`
	`}`

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 …

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

      // Prevent the withdrawal of money from the account
      public 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.balance);          // displays 0.0

s.deposit(120);
System.out.println(s.balance);          // displays 120.0

s.withdraw(20);                         // this will do nothing now
System.out.println(s.balance);          // displays 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.

At this point, we now have **SavingsAccounts** that cannot be withdrawn from and normal **BankAccounts** that can be withdrawn from. Let us 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 to $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
public 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
public 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
public void deposit(float amount) {
      int   wholeDollars = (int)(amount/100);
      super.deposit(amount + (wholeDollars * 0.50f));
}
```

or even simpler:

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

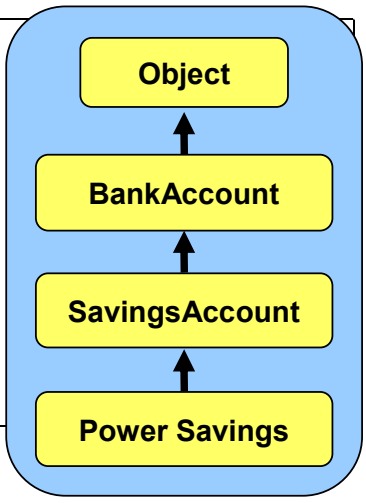
Well, we would be asking JAVA to call the **deposit()** method in ***this*** class, not the one in **BankAccount**. Furthermore, since this code is written ***inside*** the **deposit()** method, we are telling JAVA to call the method that we are actually trying to write! So the method will keep calling 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 …

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

      // Withdraw money from the account
      public void withdraw(float amount) {
          if (this.balance >= (amount + 1.25f))
              this.balance -= (amount + 1.25f);
      }
}
```



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.balance);          // displays 0.0

s.deposit(320);
System.out.println(s.balance);          // displays 321.50

s.withdraw(20);
System.out.println(s.balance);          // displays 300.25
```

Notice that the **withdraw()** method properly deducts the $1.25 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**:

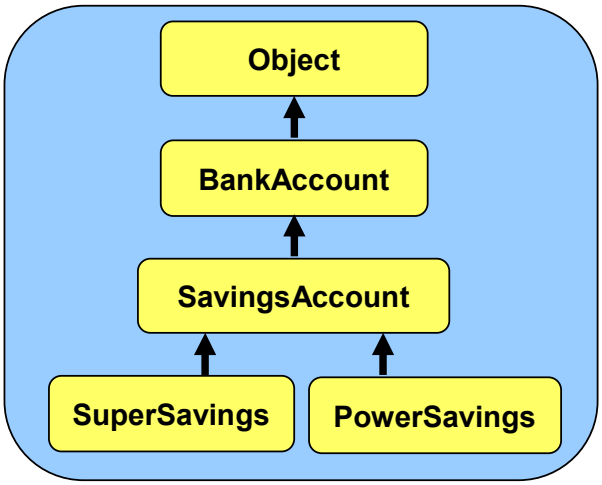
```
public class PowerSavings extends SavingsAccount {
      // Constructor to call the superclass constructor
      public PowerSavings(String aName) { super(aName); }
      public PowerSavings() { super(""); }

      // Withdraw money from the account
      public 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:

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

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

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

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

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

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

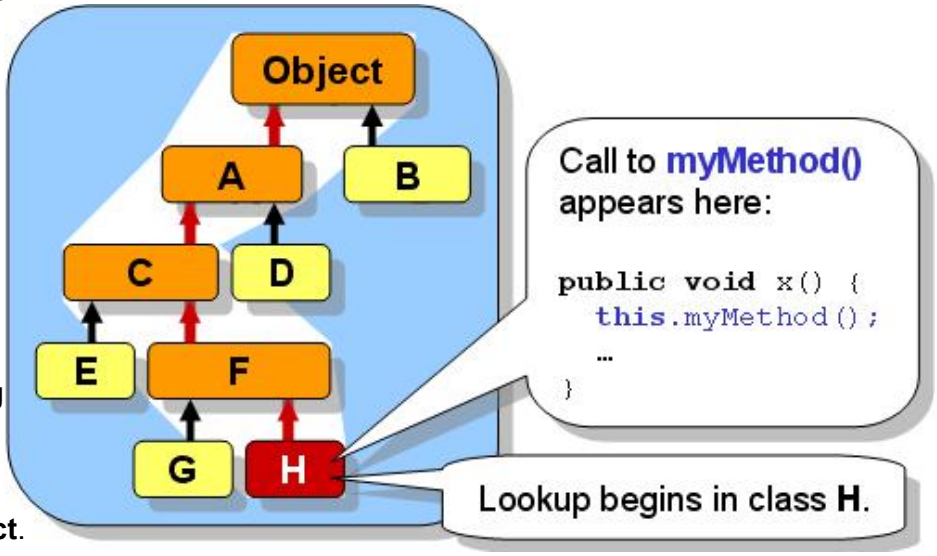
The code will work as we expect it to now, taking full advantage of inheritance.

To properly understand method calling and overriding when dealing with class hierarchies, we need to consider how JAVA "finds" a method in the class hierarchy when you try to call it. It can be confusing if there are many "overridden" methods (i.e., all with the same name and parameter lists), because we may not know which one JAVA will use. Fortunately, there is a simple way to figure this out.

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**.



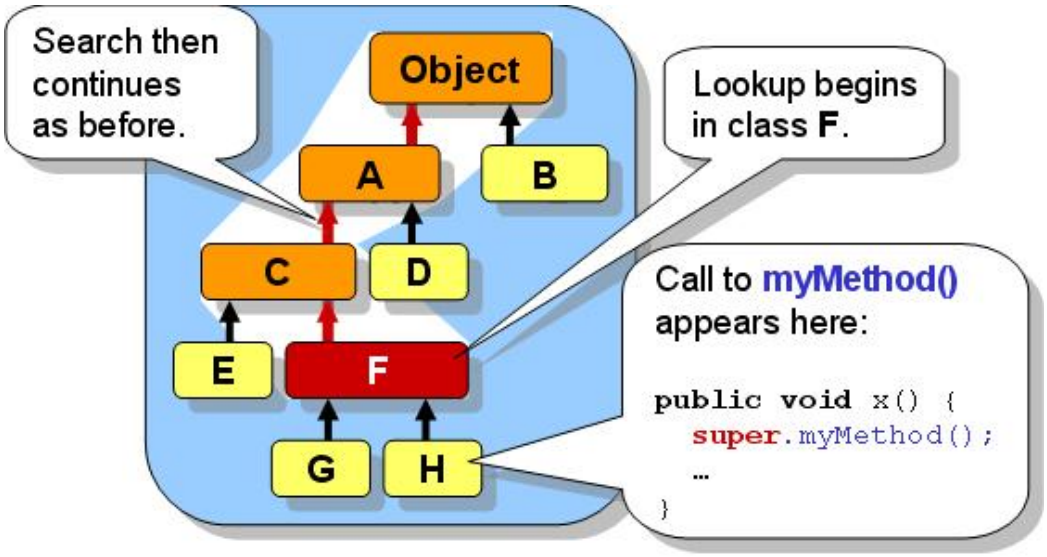
If not found at all during this search up to the **Object** class, the compiler will catch this and inform you that it cannot find method **myMethod()** for the object you are trying to sending it to:

```
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*. 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.

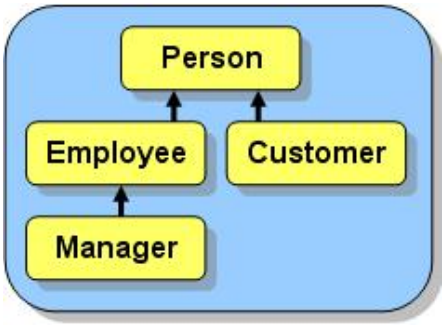
In fact if there was an implementation of **myMethod()** in the **H** class, it would not be called if we used **super**, since the search begins in the superclass, not in **this** class. So, the use of **super** merely specifies "*where the method lookup should **begin** the search*" ... nothing more.



How Are Access Modifiers Affected By Inheritance ?

It would be good to consider the effects that access modifiers have on attributes and methods within the class hierarchy. When an inherited attribute is declared as **private**, the subclasses still inherit it, but they cannot access it directly from within their own "local" code. For example, recall our previous example with **Customer**, **Manager** and **Employee** objects. Consider that all attributes are declared as **private**:

```
public class Person {
      private String   name;
      private Address  address;
      private String   phoneNumber;
}
public class Employee extends Person {
      private int      employeeNumber;
      private float    hourlyPay;
}
public class Customer extends Person {
      private String[]   itemsPurchased;
      private Date[]     purchaseHistory;
}
public class Manager extends Employee {
      private String[]       duties;
      private Employee[]     subordinates;
}
```



Now consider the following code in this method written in the **Manager** class which determines whether or not a Manager has seniority. Assume that a manager has seniority if their employee number is less than 100 and they have more than 5 employees working for them.

```
public boolean hasSeniority() {
      return  (employeeNumber < 100)  && (subordinates.length > 5);
}
```

The code will NOT compile because the code is written in the **Manager** class but the inheritted attribute employeeNumber is declared private within the **Employee** class. The subordinates attribute can be accessed without problems because it is defined in the same class as this method is written (i.e., the **Manager** class).

Since we need to access the employeeNumber attribute from the method ... how do we fix this ? There are two solutions:

(1) Write a **public getEmployeeNumber()** method in the **Employee** class and use it:

```
public class Employee extends Person {
      private int      employeeNumber;
      private float    hourlyPay;

      public int getEmployeeNumber() { return employeeNumber; }
}
```

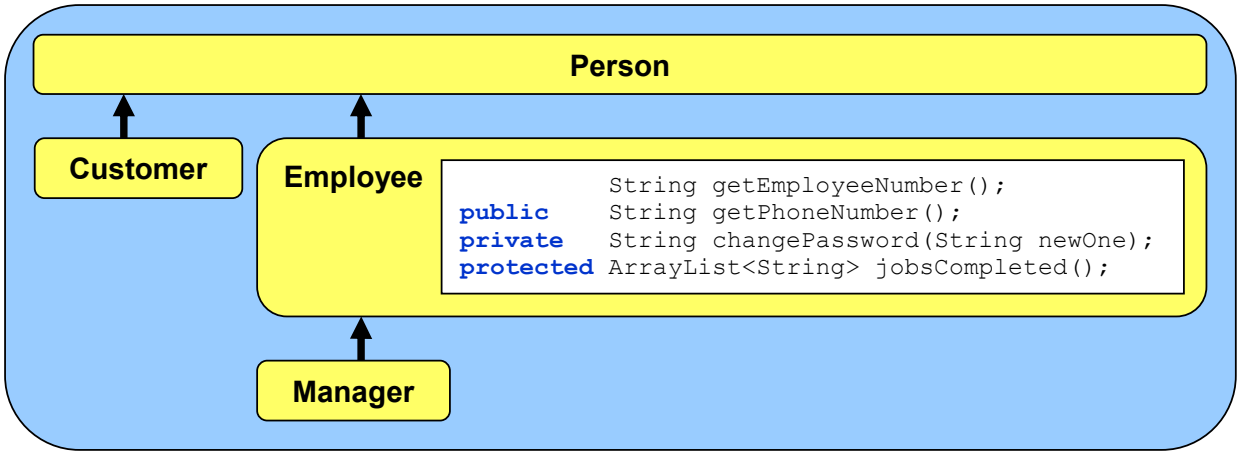
```
public class Manager extends Employee {
      private String[]      duties;
      private Employee[]    subordinates;

      public boolean hasSeniority() {
            return (getEmployeeNumber() < 100)  && (subordinates.length > 5);
      }
}
```

(2) Declare all attributes that may need to be inherited as **protected** instead of **private**. By using **protected**, all subclasses can access the attribute directly, but no other classes may.

```
public class Employee extends Person {
      protected int    employeeNumber;
      protected float  hourlyPay;
}
```

```
public class Manager extends Employee {
      private String[]      duties;
      private Employee[]    subordinates;

      public boolean hasSeniority() {
            return (employeeNumber < 100)  && (subordinates.length > 5);
      }
}
```

Now how do **private** and **protected** modifiers affect methods ? Consider four methods within the **Employee** class with various access modifiers as follows:



Now consider some code within the **Manager** class that attempts to access these methods:

```
public class Manager extends Employee  {
   public void tryThingsOut()  {
        System.out.println(this.getEmployeeNumber());     // access allowed
        System.out.println(this.getPhoneNumber());        // access allowed
        System.out.println(this.changePassword("12345678"));// compile error
        System.out.println(this.jobsCompleted());         // access allowed
    }
}
```

Notice that the only method not allowed to be accessed is the **private** method, since the **tryThingsOut()** method is written in the **Manager** class, not in **Employee**.

Consider now the **Customer** class restrictions:

```
public class Customer extends Person  {
   public void buyFrom(Employee emp)  {
        System.out.println(emp.getEmployeeNumber());        // access allowed
        System.out.println(emp.getPhoneNumber());           // access allowed
        System.out.println(emp.changePassword("12345678"));// compile error
        System.out.println(emp.jobsCompleted());            // compile error
    }
}
```

Now we can no longer call the **jobsCompleted()** method, since it has been declared **protected** and **Customer** is not a subclass of **Employee**.

There is one more "protective" keyword that can be used with methods. We can declare a method as **final** to prevent subclasses from modifying the behavior. That is, when we declare a method as being **final**, JAVA prevents anyone from *overriding* that method. Hence no subclasses can have a method with that same name and signature:

```
public final void withdraw(float amount) {
    ...
}
```

Why would we want to do this ? Perhaps the behavior defined in the method is very critical and overriding this behavior "improperly" may cause problems with the rest of the program.

Restricting Class Access

In regards to class definitions, we are also allowed to indicate either *default* or **public** access to the class. So far, all of our classes have had **public** access, but we can have default access by leaving off the keyword **public**:

```
public class Manager {  // public access from classes anywhere
    ...
}
```

```
class Employee {  // default access from classes within package/folder
    ...
}
```

Interestingly, we can also declare a class as **final**. This means that it CANNOT have subclasses:

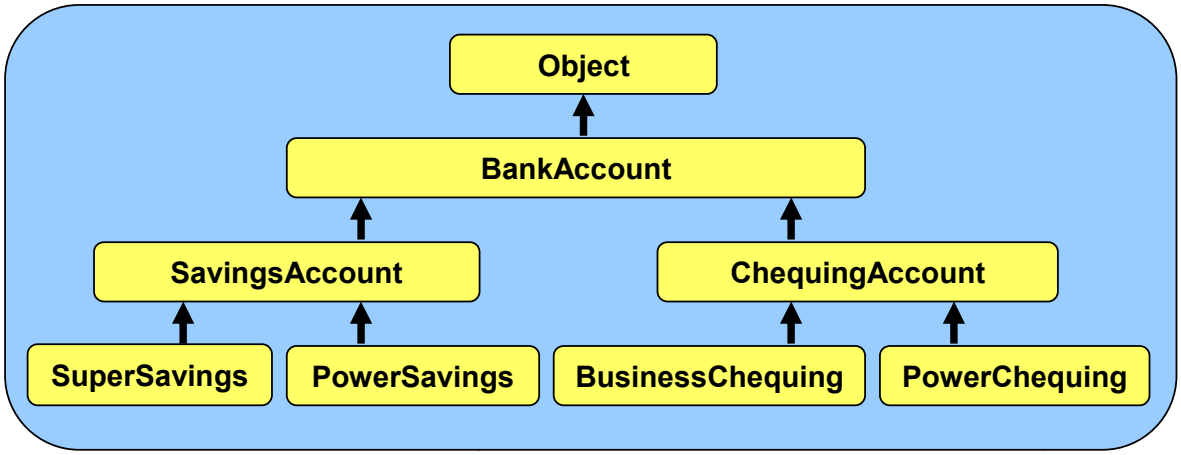
```
public final class Manager {
    ...
}
```

Why would we want to do this ? Perhaps the class has very weird code that the author does not want you to inherit ... maybe because it is too complicated and may easily be misused. Many of the JAVA classes (e.g., **ArrayList**) are declared as **final** which means that we cannot make any subclasses of them. It is a kind of security issue to prevent us from "messing up" the way those classes are meant to be used. It's a shame, because often we would like to have special types of **ArrayLists** and other similar objects.

4.3 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:

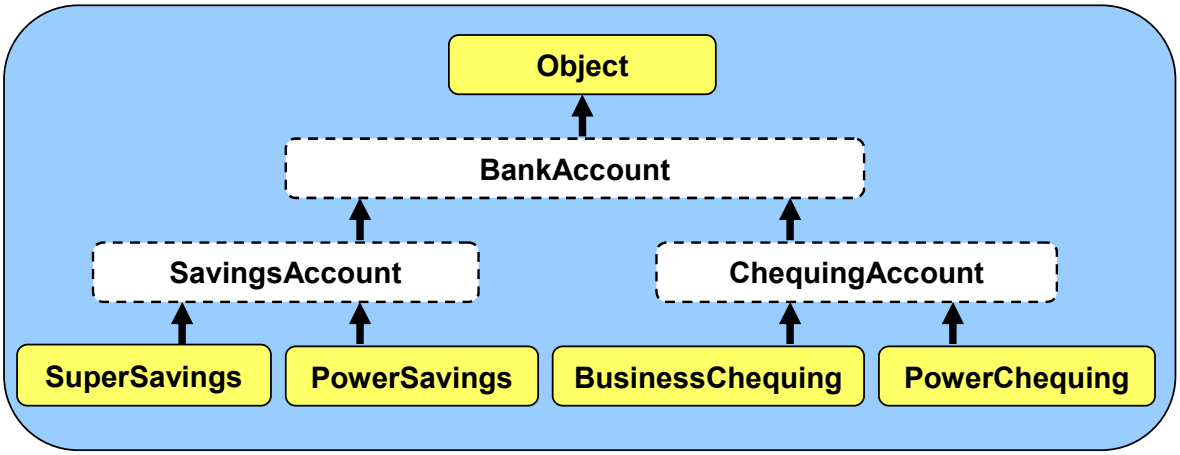
```
account1 = new SuperSavings(…);
account2 = new PowerSavings(…);
account3 = new BusinessChequing(…);
account4 = new PowerChequing(…);
```

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 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:

```
public abstract class BankAccount {
    ...
}
```

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

```
public 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 ?

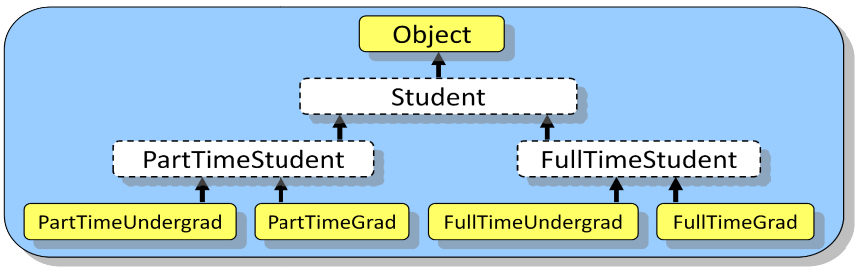
Well ... why did we create the **BankAccount** and **SavingsAccount** classes in the first place ? Inheritance was the key reason. These classes still contain the common attributes and shared behavior for all of their subclasses. The **BankAccount** class, for example, contains the 3 instance variables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

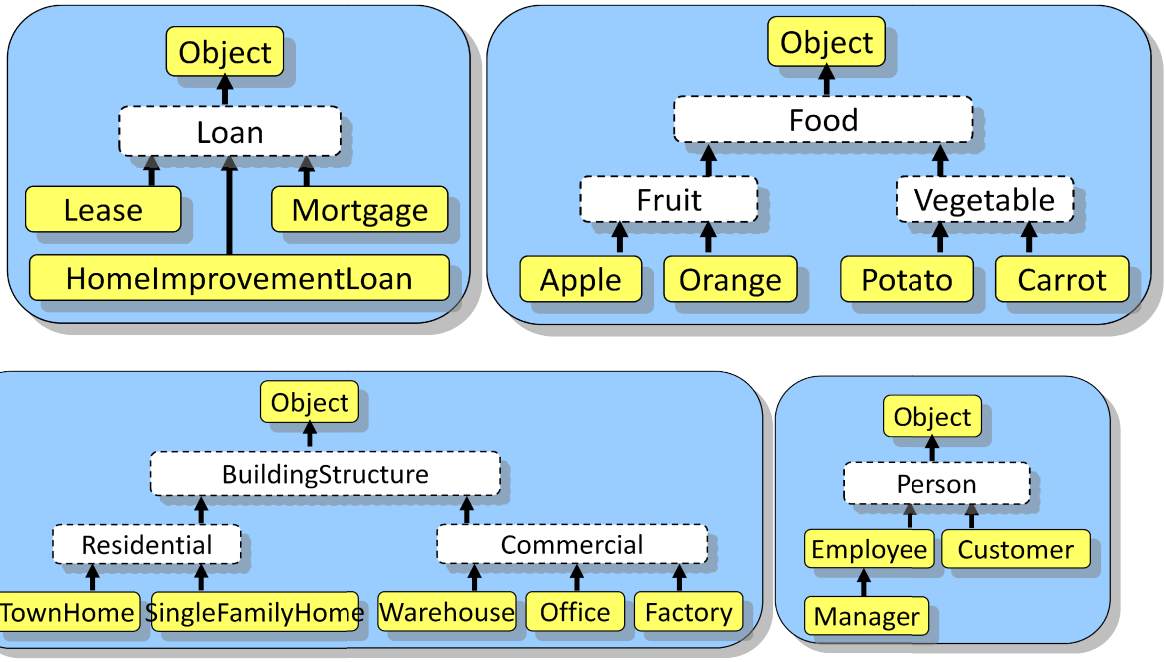
Also, the **SavingsAccount**, for example, contains the **deposit()** method that is shared between **SuperSavings** and **PowerSavings**. Hence, you can see that even though a class may be declared as **abstract** it is still useful and important in keeping our code organized properly in our class hierarchy. 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 with no code** for which all concrete subclasses are **forced** to implement the method.*

So, an abstract method 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:

```
public abstract void deposit(float amount);
public 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 **;**

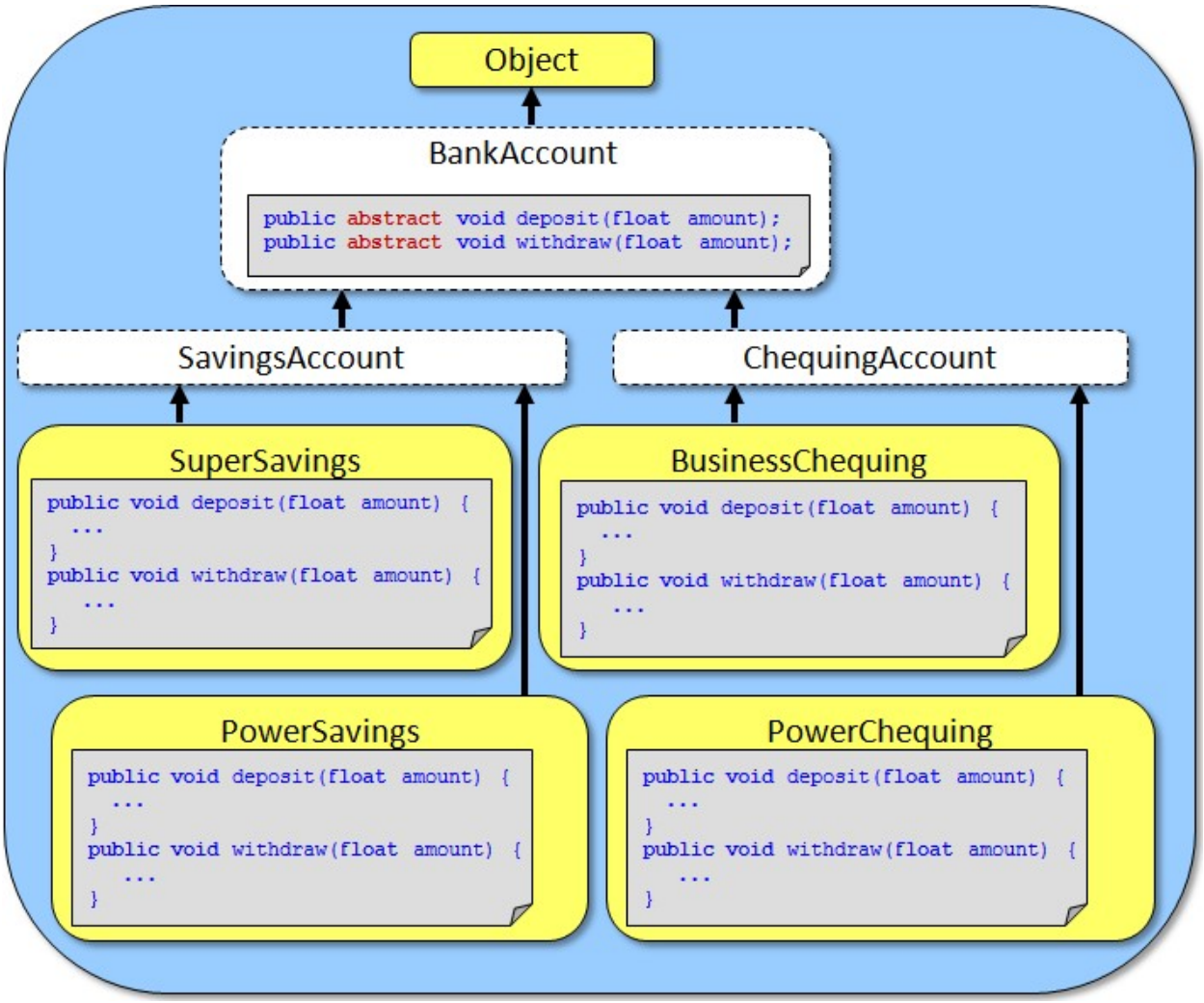
At this point you should be wondering: "**Why would any sane person would write a method that has no code in it ?**". That is certainly a reasonable question since, after all, methods are called so that we can evaluate the code that is in them.

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.

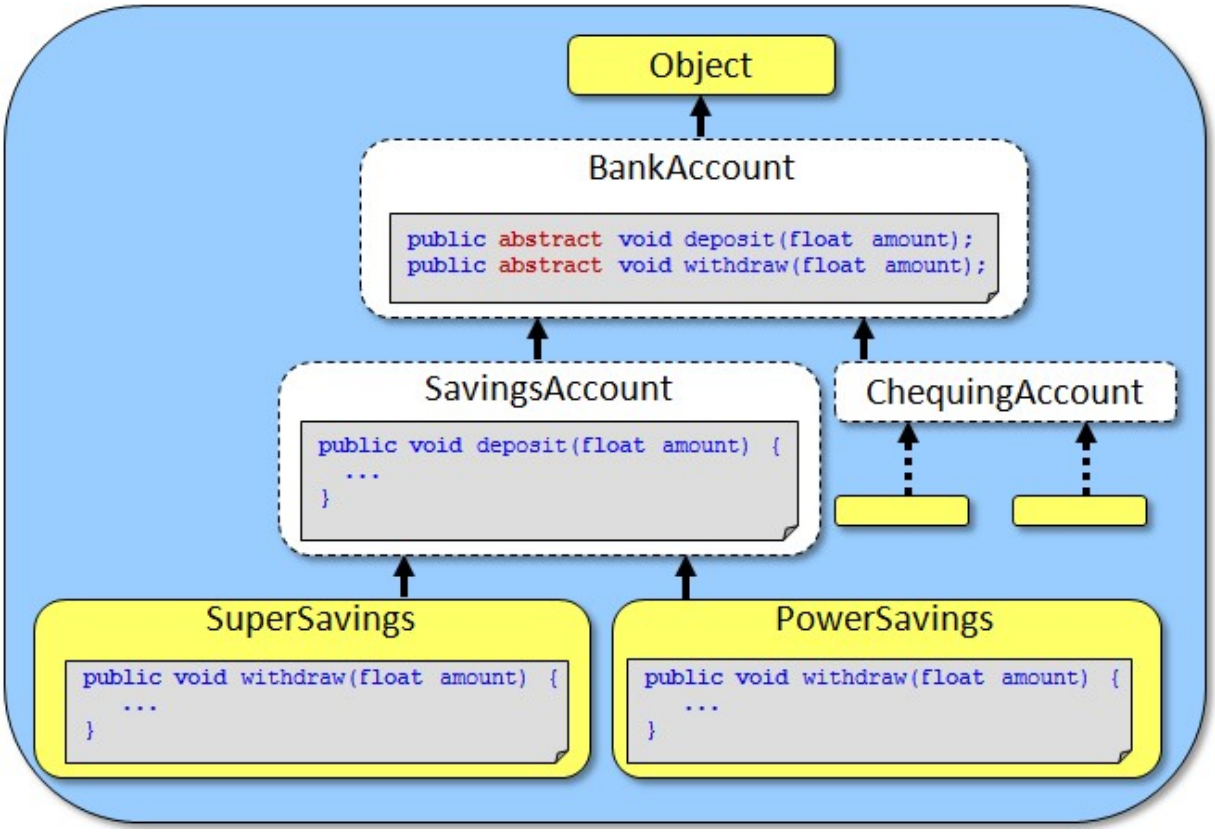
For example, if we make **deposit(float** amount**)** and **withdraw(float** amount**)** methods **abstract** in the **BankAccount** class, then, all of its concrete subclasses (**SuperSavings**, **PowerSavings**, **BusinessChequing** and **PowerChequing**) would be forced to implement those methods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

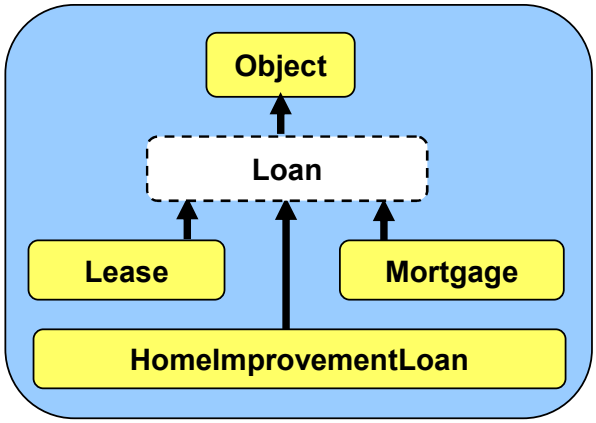


Only **abstract** classes are allowed to have such abstract methods. However, as you know, an **abstract** class may have regular methods as well.

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:

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

    public 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 ?:

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

4.4 JAVA Interfaces

Inheritance allows all classes along the same path in the class hierarchy to share attributes and behaviors. The structure of the class hierarchy helps to identify common behavior that subclasses have with their superclasses. How though, would we define (and perhaps *force*) common behavior between seemingly *unrelated* classes in different parts of the class hierarchy ?

There is a mechanism in JAVA for doing this:

> *An **interface** is a specification (i.e., a list) of a set of methods such that any classes implementing the interface are forced to write these methods.*

Using an interface 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 such a list of methods as if we were defining a new class, except that we use the keyword **interface** instead of **class**:

```
public 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**.

Here is an example of an interface that defines a **Loanable** object:

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

The methods themselves are defined like **abstract** methods, but without the word **abstract**. For comparison purposes, recall the similar **abstract** class called **Loan** with abstract methods:

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

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

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

- both define three similar methods with no code.
- like abstract classes, we cannot create instances of interfaces. So, we cannot do the following anywhere in our code: `new Loan()` nor `new Loanable()`

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.
- All methods in an **interface** must be declared **public**

Since interfaces are defined by themselves in their own files (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.

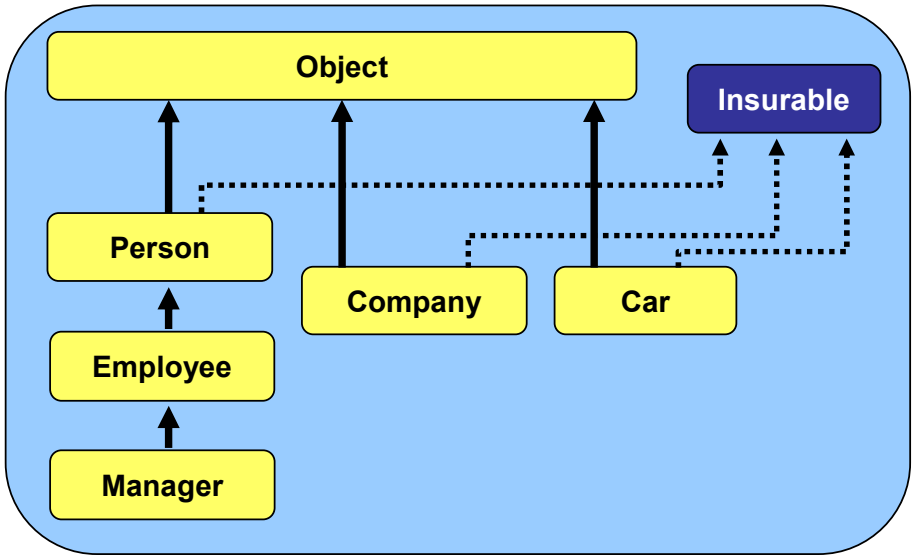
Consider defining an interface called **Insurable** that defined the common behavior that all insurable objects MUST have as follows:

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

The code above would need to be saved and compiled before we can use it.

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:

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

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

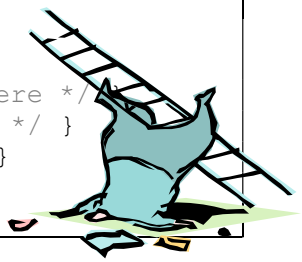
```
public class Car implements Insurable {
    ...
}
```

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

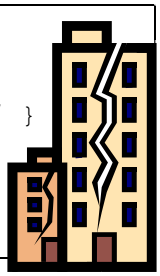
methods required for insurable items (i.e., **getPolicyNumber()**, **getCoverageAmount()**, **calculatePremium()** and **getExpiryDate()**). So then, for each of the implementing classes, we must go and write the code for those methods:

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

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

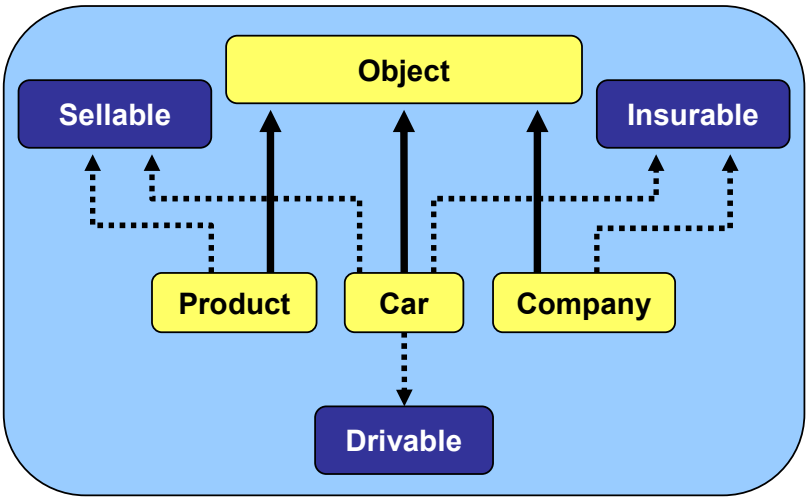


```
public class Company implements Insurable {
    //...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium(int days) { /* write code here */ }
    public java.util.Date  getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /*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



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.

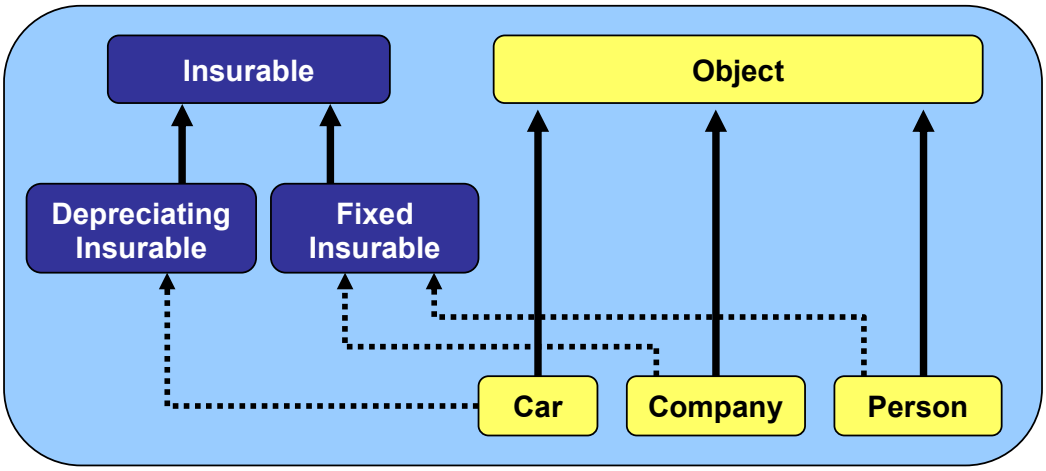
Interestingly, a class may implement more than one interface:



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:

```
public 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:

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

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

```
public 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 when we discuss user interfaces.

4.5 Polymorphism

Recall that we can convert (or type-cast) primitives to convert a value from one type to another:

```
(int)871.34354;    // results in 871
(char)65;          // results in 'A'
(long)453;         // results in 453L
```

Some type-casting is done automatically by JAVA when we assign a value of one particular type to a variable of a different type. However, we can also explicitly type-cast in order to simplify the data (e.g., from **float** to **int**) or for display purposes (e.g., from **byte** to **char**).

In JAVA, we can also type-cast **objects** from one type to another type. However, type-casting objects is ***different*** from type-casting primitives in that the objects are ***not converted or modified*** in any way. Instead, when we type-cast an object variable, it is simply restricted with respect to the kinds of behaviors that it is capable of doing from then on in our program.

Why would we want to do type-casting if all that we are doing is restricting the object in some way. Would it not be better (i.e., more flexible) to simply allow the object's methods to be used at any time ? These are valid questions. However, there are reasons for type-casting.

Perhaps the main advantage of type-casting is that it allows for:

Polymorphism *is the ability to use the same behavior for objects of different types.*

That is, it allows different objects to respond to the exact "same" methods. The result is that we have much less to remember when we go to use the object. That is, by using polymorphism, we just need to understand a few commonly used methods that all these objects understand. For example:

- We can ask all **Person** objects what their **name** is. This is independent as to whether or not they are instances of **Employees**, **Managers**, **Customers** etc...
- We can deposit to any **BankAccount**, independent of its type.

And so … by treating an object more generally (i.e., type-casting it), we are simplifying the way that we will use the object by restricting its usage to a few well understood methods. As a result, our code becomes

- easier to understand
- more intuitive and
- quicker to write since the programmer does not need to remember as many methods.

It is important to understand the type-casting of objects because JAVA often type-casts objects automatically. Therefore, we must understand *how* to type-cast and *when* it is done automatically. The type-casting of objects is done the same way (i.e., with the round brackets) as with primitives. Here are a few examples:

```
p = (Person)anEmployee;
c = (Customer)anArray[i];
b = (SavingsAccount)aBankAccount;
```

Notice that there is an object type (i.e., class name) within the round brackets/parentheses.

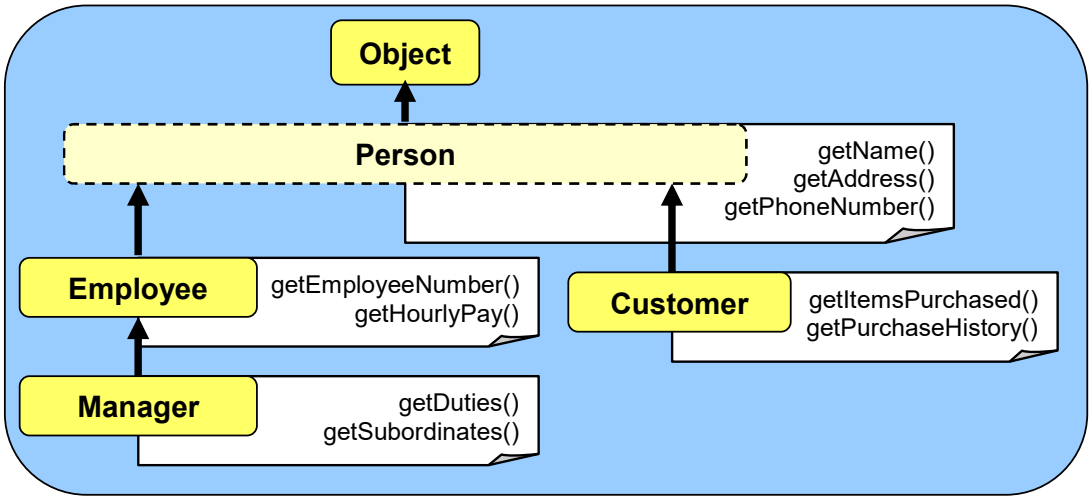
When we type-cast an object to another type we are not modifying it in any way. Rather, we are simply causing the object to be **"treated"** more generally from then on in the program. As a result, the object will then be less flexible in that we can no longer call some of the methods that we used to call on it. In a way, we are ignoring some of the behavior that is available to the object.

This may sound strange, but we do this in real life. Let us consider a couple of examples.

Consider meeting your professor with his family outside of class, perhaps at a local shopping mall. Likely, you would "treat" your professor as a general/normal **Person** ... not as your "professor". So, you might ask him questions that you would ask anyone such as: "*Is this your family?*" or "*What are you shopping for today?*". However, you would likely not ask him a question like "What kind of questions will be on the final exam?" and hopefully you would not pull out a laptop and ask him to help you debug the code on your assignment. So, in a sense, you have type-casted the **Professor** to a more general **Person** object by restricting the available behaviors to those that are applicable to more general people, avoiding any professor-specific behavior.

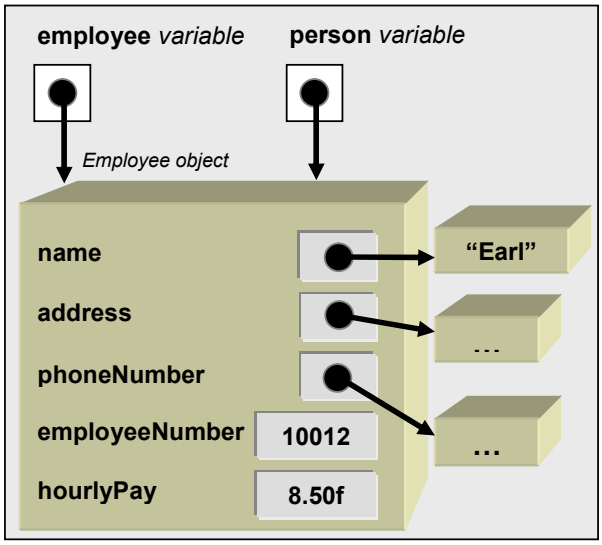
As another example, consider an **Apple** … normally you may **polish**, **peel** or **eat** it ... but in a food fight, you may type-cast (i.e., treat) your apple as a general ***throwable*** projectile. Then, the apple takes on different behavior such as **throw**, **catch**, **splatter**, etc... The fact is ... it is still an **Apple**, but it is being *treated* differently. You may even type-cast other objects to be projectiles such as grapes, sandwiches, pineapples (ouch), chairs, etc...

Now let us look at a real coding example. Consider the following class hierarchy of **Employee**, **Person**, **Manager** and **Customer** objects with some instance methods belonging to each class as shown:



Consider what happens when we create a single **Employee** object and then type-cast it to a **Person**. Take note of the methods that are available for use and those which will not compile. Note that we create 2 variables, yet both point to the same object …

```
Person     person;
Employee   employee;

employee = new Employee("Earl");
employee.getName();
employee.getAddress();
employee.getPhoneNumber();
employee.getEmployeeNumber();
employee.getHourlyPay();

// now treat Earl like a person
person = (Person)employee;
person.getName();
person.getAddress();
person.getPhoneNumber();

// these two will not compile
person.getEmployeeNumber();
person.getHourlyPay();

// type-cast back and all is ok
((Employee)person).getEmployeeNumber();
((Employee)person).getHourlyPay();
```



You will notice that once the type-cast to **(Person)** occurs, we are no longer able to use the **getEmployeeNumber()** and **getHourlyPay()** methods since they are **Employee**-specific

methods and we are now treating Earl as simply a **Person**. However, the **person** variable is still pointing to Earl … the exact same object.

When we type-cast the **person** variable back to **(Employee)** again, and then try the same two methods, they work fine because we are now treating Earl as an **Employee** again.

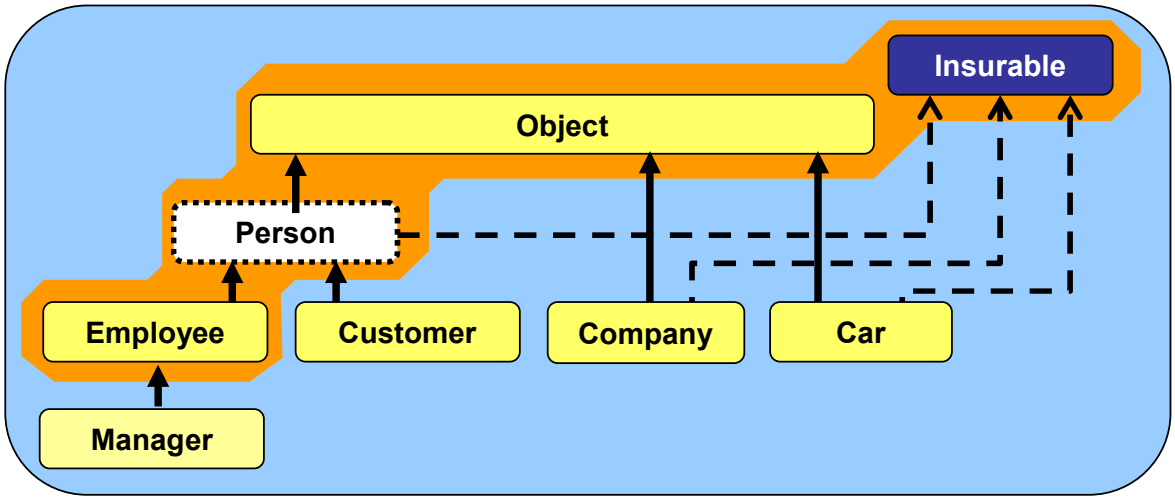
Notice what we are **not** able to do:

```
Employee   employee;
Manager    manager;
Customer   customer;

employee = new Employee("Earl");
manager = (Manager)employee;     // Type-cast is not allowed
customer = (Customer)employee;   // Type-cast is not allowed
```

We are only allowed to use class type-casting to generalize an object. Therefore we can only type-cast to classes up the hierarchy (e.g., **Person** and **Object**) but not down the hierarchy (e.g., **Manager**) or across the hierarchy (e.g., **Customer**) from the original object class (e.g., **Employee**). In summary, objects may **ONLY** be type-casted to:

- a type which is one of its **superclasses**
- an **interface** which the class implements
- or back to their own class again

In the following example, an **Employee** object can *only* be type-casted to (or stored in a variable of type) **Employee**, **Person**, **Object** or **Insurable**:



Attempts to type-cast to anything else will generate a **ClassCastException**. So **Employees** CANNOT be type-casted to **Manager**, **Customer, Company** or **Car**. Such restrictions make sense, after all, why would we "treat" a **Manager** as a **Company** or a **Car.**

Some coding advantages arise through *implicit* or *automatic* type-casting. Sometimes JAVA will automatically type-cast an object, even if we do not explicitly do so with the brackets **()**.

There are two main situations in which automatic type-casting occurs:

1. when we assign an object to a variable with a more general type:

```
Person    person;
Employee  employee;

employee = new Employee("Earl");
person = employee;    // same as person = (Person)employee;
```

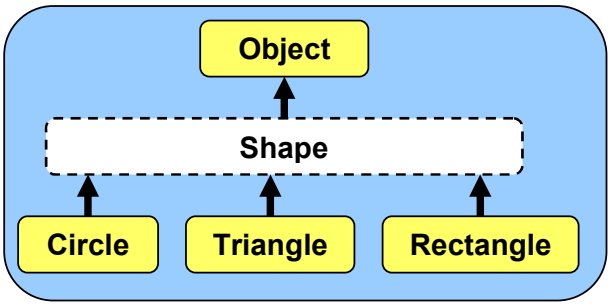
2. when we pass in the object as a parameter to a method which has a more general type:

```
Employee  employee;

employee = new Employee("Earl");
doStandardHiringProcess(employee);
...
```

```
public void doStandardHiringProcess(Person  p) {
   // employee object is type-casted to Person upon entering method
   ...
}
```

In both cases, you should be aware that an automatic type-cast has taken place. In fact, it usually does not matter if you "know" that the type-casting is taking place, because the compiler will tell you. However, it tells you this by means of a compile error … which is somewhat unpleasant, as you well know. Also, sometimes the compiler message is not straightforward to understand.

Let us now look at a simple example to see how much we can reduce our code through the use of automatic type-casting. Consider a hierarchy of shape-related objects as shown here. We can create a **Circle**, a **Triangle** and a **Rectangle** and all three can be stored into a variable of type **Shape**:




```
Shape      s;
Circle     c = new Circle(20);
Triangle   t = new Triangle(10, 20, 30);
Rectangle  r = new Rectangle(10, 10, 20, 20);
s = c;     // s points to object c
s = t;     // s points to object t
s = r;     // s points to object r
```

Notice that we did not make any explicit type-cast to **Shape** (although we could have done so). Here we simply re-assigned variable **s** to have three different values corresponding to three different types of objects. The example code itself is pointless, but it helps us to see how we can use automatic type-casting.

Assume now that we want to draw a shape and that the **Circle**, **Triangle** and **Rectangle** classes all have an appropriate method for drawing themselves called **draw()**:

```
public class Circle extends Shape {
    ...
    public void draw() { ... }
}
```

```
public class Triangle extends Shape {
    ...
    public void draw() { ... }
}
```

```
public class Rectangle extends Shape {
    ...
    public void draw() { ... }
}
```

Consider now our **Shape** variable **s** which can hold any kind of shape:

```
Shape   s = ...;
```

At any given time, we may not know exactly which kind of shape is currently stored in the **Shape** variable **s**. How then do we know which **draw()** method to call ? Well, we could check the type of the object, perhaps with the **instanceof** keyword (which returns a boolean indicating whether or not the object is an instance of a particular class) and then use some **if** statements as follows:

```
if (s instanceof Circle)
    s.draw();

if (s instanceof Triangle)
    s.draw();

if (s instanceof Rectangle)
    s.draw();
```

However, looking at the code, it is clear that regardless of the type of shape we have, we just need to call **draw()**. Since we called all of the methods **draw()**, this is an example of polymorphism … that is … all shape objects understand the **draw()** method. For this to compile though, there should also be a **draw()** method defined in the **Shape** class, which may be blank.

As a result, because of polymorphism and the explicit type-cast, we don't even need the **IF** statements. Our code can be simplified to:

```
s.draw();
```

Incredible!!! What a reduction in code! But why does this work ? How does JAVA know which **draw()** method to call ? Well, remember, whatever we store in the **Shape** variable **s** does not change its type. The compiler will look at the kind of object that we put in there and call the appropriate method accordingly by starting its method lookup in the class corresponding to that object type (i.e., either **Circle**, **Triangle** or **Rectangle**, depending on what was stored in **s**). As you can see, polymorphism can be quite powerful.

Now consider a **Pen** object which is capable of drawing shapes. We would like to use code that looks something like this:

```
Pen  aPen = new Pen();

aPen.draw(aCircle);
aPen.draw(aTriangle);
aPen.draw(aRectangle);
```

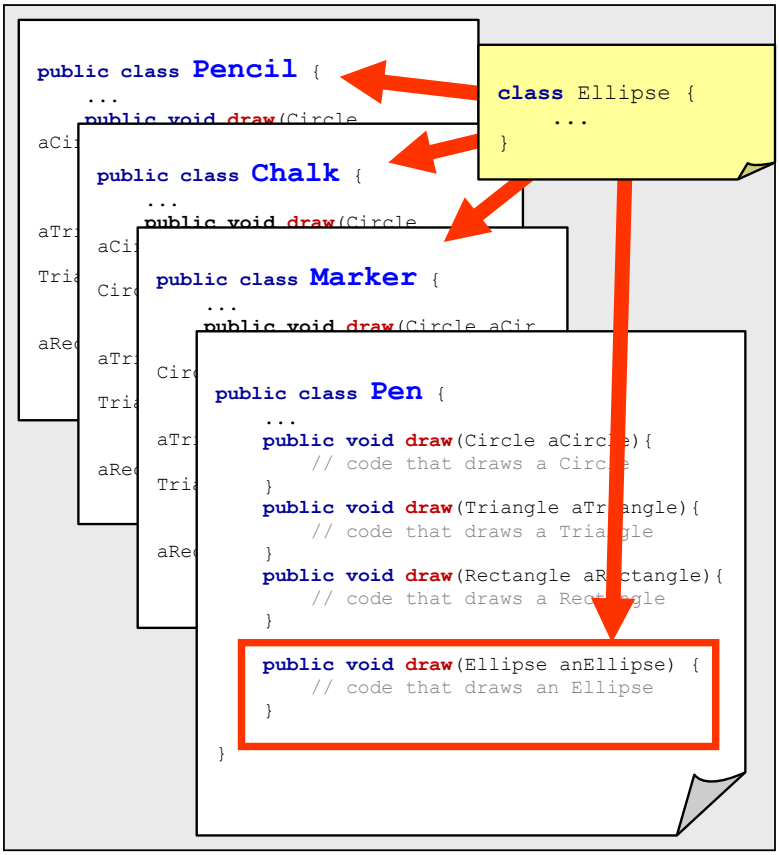
However, this is not so straight forward. We would have to define a **draw()** method in the **Pen** class for each kind of shape in order to satisfy the compiler with regards to the particular type of the parameter:

```
public class Pen {
    ...
    public void draw(Circle aCircle) {
        // code that draws a Circle
    }
    public void draw(Triangle aTriangle) {
        // code that draws a Triangle
    }
    public void draw(Rectangle aRectangle) {
        // code that draws a Rectangle
    }
}
```

Since the drawing code is likely different for all 3 shapes we will need the 3 different pieces of code to do the drawing. However, all of the shape-drawing code must appear here in the **Pen** class. This is somewhat intuitive in regards to real life, since **Pen's** draw shapes.

However, if we had other drawing classes such as **Pencil**, **Marker** or **Chalk**, we would need to go to all these classes and insert shape-specific code for each kind of shape. Even worse, if we wanted to add shapes (e.g., **Ellipse**, **Diamond**, **Parallelogram**, **Rhombus**, etc..) then we would have to go to the **Pen**, **Pencil**, **Marker** and **Chalk** classes to add the appropriate shape-drawing code.

This is quite terrible since our code is not modular … the adding of one simple **Shape** class would require us to recompile 4 other classes.



There must be a better way to do this! The answer is to use a technique known as ***double-dispatching***. When we call a method in JAVA, this is the same notion as *sending a message* to the object. The idea behind double-dispatching is to *dispatch* a JAVA message two times. Through double-dispatching, we force a second message to be sent (i.e., we call another method) in order to accomplish the task.

Before we do the double-dispatch, we need to adjust our code a little. We can simplify the **draw()** methods in the **Pen**, **Pencil**, **Marker** and **Chalk** classes by combining them all in one method. The new method will take a single parameter of type **Shape**. Hence, through type-casting, we can pass in any subclass of **Shape** to the method. Here is the code …

```
public class Pen {
    ...
    public void draw(Shape anyShape) {
        if (anyShape instanceof Circle)
            // Do the drawing for circles
        if (anyShape instanceof Triangle)
            // Do the drawing for triangles
        if (anyShape instanceof Rectangle)
            // Do the drawing for rectangles
    }
}
```

At this point, we still have to decide how to draw the different **Shapes**. So then when new **Shapes** are added, we still need to come into the **Pen** class and make changes. However, we can correct this problem by shifting the drawing responsibility to the individual shapes themselves, as opposed to it being the **Pen's** responsibility. This "shifting" (or flipping) of responsibility is where the notion of ***double-dispatching*** comes in. It is similar to the expression "passing-the-buck" in English. In other words, we are saying: ***"I'm not going to do it ... you do it yourself"***.

We perform double-dispatching by making a method in each of the specific **Shape** subclasses that allows the shape to draw itself using a given **Pen** object:

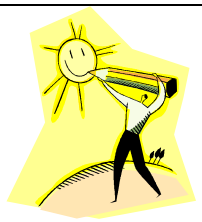
```
public class Circle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

```
public class Triangle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

```
public class Rectangle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

Then, we do the double-dispatch itself by calling the **drawWith()** method from the **Pen** class:

```
public class Pen {
    ...
    public void draw(Shape aShape) {
        aShape.drawWith(this);
    }
}
```

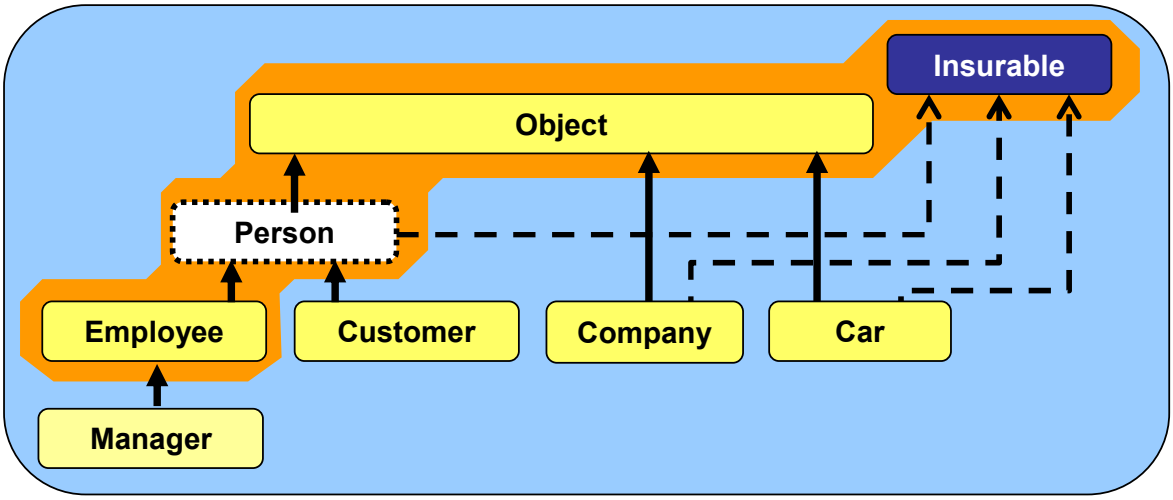


Notice that the code is incredibly simple. When the **Pen** is asked to draw a **Shape**, it basically says: "No way! Let the shape draw itself using ME!". That is the second message call, which itself does the real drawing work. We would write a similar one-line method in the **Pencil**, **Chalk** and **Marker** classes. In order for this to compile, you must also have a **drawWith(Pen aPen)** method declared in class **Shape** even if that method does nothing.

Do you see the tremendous advantages here ? Regardless of the kind of **Shape** that we may add in the future, we NEVER have to go into the **Pen**, **Pencil**, **Marker** or **Chalk** classes to make changes. This code remains intact. Instead, we simply write a **drawWith()** method in the new **Shape** class to do the drawing of itself. And who would know better how to draw the

shape than itself. The code is much more modular and has a nice clean separation. Furthermore, the code is logical and easy to understand.

Type-casting also provides advantages when multiple unrelated classes implement the same interface. Objects can be type-casted to an interface type, provided that the class implements that interface. In the hierarchy below, we can type-cast any instances of **Car**, **Company**, **Customer**, **Employee** or **Manager** to **Insurable**.



Assume that **Insurable** has a method defined called **getPolicyNumber()** and that the **Car** class has a **getMileage()** method. Notice the type-casting as follows:

```
Car        jetta = new Car();
Insurable  item = (Insurable)jetta;

item.getPolicyNumber();        // OK since Insurable
jetta.getMileage();            // OK (assuming it is a Car method)
item.getMileage();             // Compile Error
((Car)item).getMileage();      // OK now
```

Notice the compile error when calling **getMileage()** on **item**. Even though **item** is actually a **Car** object, it has been type-casted to **Insurable**, and so only methods that are defined in the **Insurable** interface can be used on it.

What is the advantage of type-casting to an interface ? Well, we can treat "seemingly unrelated" objects the same way. This is often useful when we have a collection of such items.

Consider an Array of a variety of **Insurable** items and then trying to list all of the policies and totaling the amounts of all the policies:

```
float           total = 0;
Insurable[]     insurableItems;

insurableItems = new Insurable[5];
insurableItems[0] = new Car("Porsche", "Carerra", "Red", 340);
insurableItems[1] = new Customer("Guy Rich");
insurableItems[2] = new Company("Elmo's Edibles", 2009);
insurableItems[3] = new Employee("Jim Socks");
insurableItems[4] = new Manager("Tim Burr");

System.out.println("Here are the policies:");
for (int i=0; i<insurableItems.length; i++) {
    System.out.println("  " + insurableItems[i].getPolicyNumber());
    total += insurableItems[i].getPolicyAmount();
}
System.out.println("Total policies amount is $" + total);
```

In the above example, all 5 unique objects are automatically type-casted to **Insurable** when added to the array. Then when listing the policies, we simply use the common **getPolicyNumber()** method (which must be defined in **Insurable** and implemented by all the classes). Similarly, we total all the policy amounts by using the common **getPolicyAmount()** method.

What would the code look like without having the **Insurable** interface ? Well, in order to store the items in the same array we would still need to know what they have in common. Without the **Insurable** interface, the only other thing that all the objects have in common is that they are subclasses of **Object**. So we would have to make an **Object[5]** array of general objects: `Object[] insurableItems = new Object[5];`

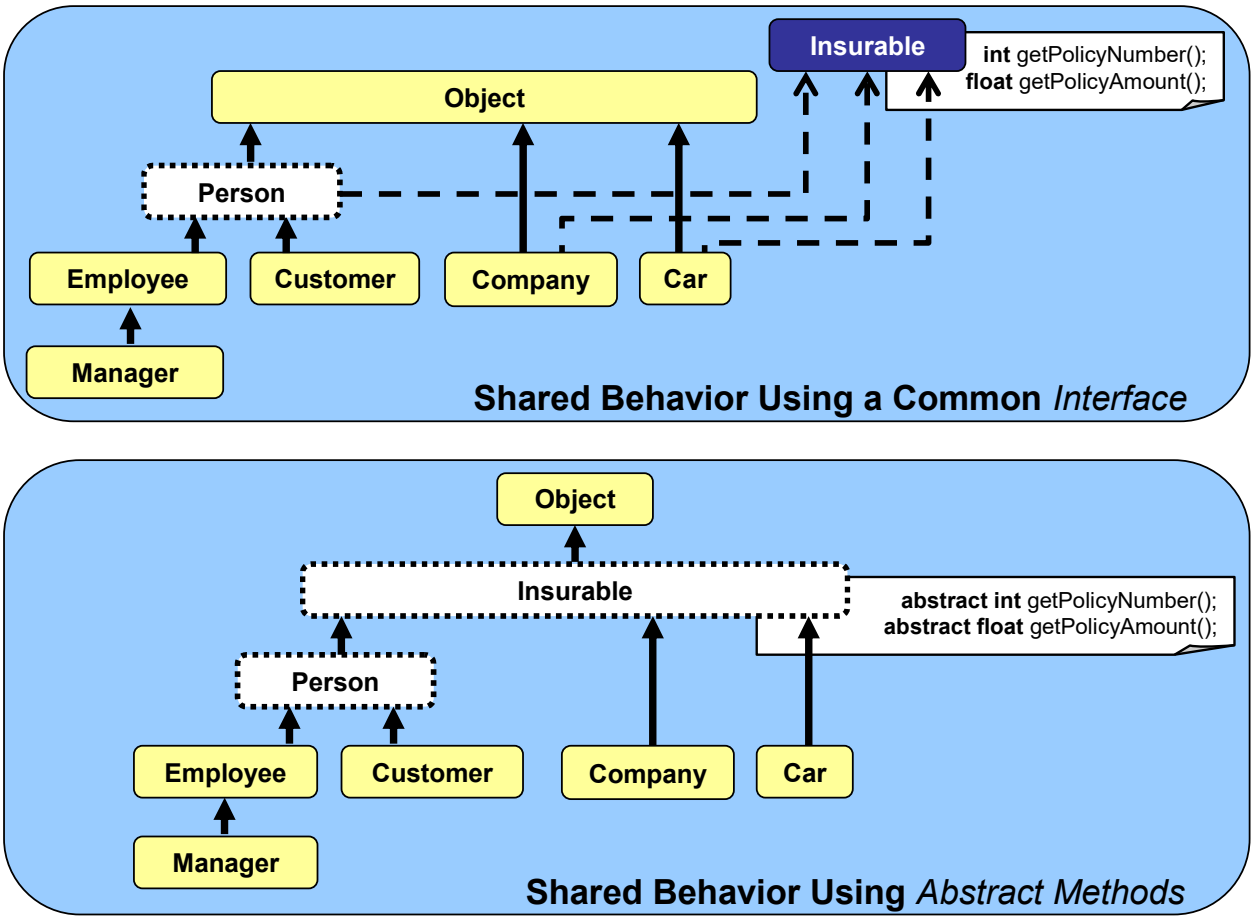
Once we make these changes, then the compiler will prevent us from calling the **getPolicyNumber()** or **getPolicyAmount()** methods because it assumes that the **item** extracted in the FOR loop is a general **Object** … but general objects do not have such methods. Therefore, we would be forced to check the type of every object, beforehand … implying that we knew all the different types that would ever be placed in the array. Our code would be longer, more complicated, messier and non-modular:

```
...
for (int i=0; i<insurableItems.length; i++) {
    if (insurableItems[i] instanceof Car) {
        System.out.println("  " + ((Car)insurableItems[i]).getPolicyNumber());
        total += ((Car)insurableItems[i]).getPolicyAmount();
    }
    else if (insurableItems[i] instanceof Employee) {
        System.out.println("  " + ((Employee)insurableItems[i]).getPolicyNumber());
        total += ((Employee)insurableItems[i]).getPolicyAmount();
    }
    else if (...)

    ...
}
```

Of course, an alternative to using the shared interface would be to have all insurable objects extend (i.e., inherit from) a common abstract class, perhaps called **Insurable** as well. We could then define the **getPolicyNumber()** and **getPolicyAmount()** methods as **abstract** methods, forcing all subclasses to implement them. Then, we could use the same identical code that worked with the **Insurable** interface.

However, the big disadvantage of doing things this way, is that we are restricting the inheritance of **Insurable** objects to be insurable-related. That means, we cannot take advantage of other kinds of inherited attributes and behaviors.

Here is a diagram showing how we could get such shared behavior either with interfaces or with abstract methods …

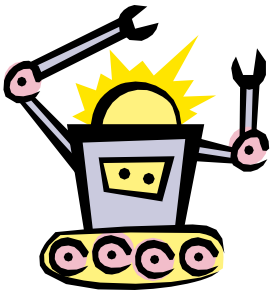


As another more tangible example, consider defining a **Controllable** interface for objects that can be controlled via remote control. The interface may look as follows:

```
public interface Controllable {
    public void  turnLeft();
    public void  turnRight();
    public void  moveForward();
    public void  moveBackward();
}
```

Now, consider a **Robot** object which is **Controllable** and implements this interface:

```
public class Robot implements Controllable {
    private int         batteryLevel;
    private Behavior[]  behaviors;

    // These are the Controllable-related methods
    public void turnLeft() { ... }
    public void turnRight() { ... }
    public void moveForward() { ... }
    public void moveBackward() { ... }

    // There will likely also be some other methods
    // which are robot-specific
    public Behavior computeDesiredBehavior() { ... }
    public int readSensor(Sensor x) { ... }
     ...
}
```



Now, what about a **ToyCar**, or even a **Lawnmower** ? We can implement the **Controllable** interface for each of these as well. In fact, suppose that we want to set up a handheld remote control for **Controllable** objects. We can then treat all of the objects (**Robots**, **ToyCars**, **Lawnmowers**, etc...) as a single type of object ... a **Controllable** object:

```
public class RemoteControl {
    private Controllable      machine;

    public RemoteControl(Controllable  m) {
        machine = m;
    }

    public void handleButtonPress(int buttonNumber) {
        if (buttonNumber == 1)
            m.moveForward();
        else if (buttonNumber == 2)
            m.moveBackward();
        else if (buttonNumber == 3)
            m.turnLeft();
        else
            m.turnRight();
    }
    ...
}
```

Notice that the remote control constructor is supplied with any object that is of type **Controllable** (i.e., a **Robot**, **ToyCar**, **Lawnmower**, etc..) Therefore, as can be seen in the **handleButtonPress()** method, the code for controlling the machine from the remote is independent of the type of object being controlled.

This is a nice clean separation of code in that any new **Controllable** object that is developed in the future can be controlled by this **RemoteControl** object. The programmer would not need to make any changes to the **RemoteControl** class code whatsoever:

```
ToyPlane    aPlane = new ToyPlane();
ToyBoat     aBoat = new ToyBoat();

RemoteControl     planeRemote = new RemoteControl(aPlane);
RemoteControl     boatRemote = new RemoteControl(aBoat);
```