Chapter 9

Proper Coding Style

What is in This Chapter ?

In this chapter, we will discuss various ways to improve the code that we write. We begin by discussing how we can simplify the way others will use our objects and we will find ways to secure the attributes of our object by using access modifiers such as **public**, **private** and **protected**. We will then discuss the use of **get** and **set** methods to access and modify our objects in a safe way. The chapter then discuses ways to simplify our constructors as well as reducing clutter in our code all by using or removing the **this** keyword. We then discuss in detail how we can **type-cast** objects into more general types as a means of simplifying our code greatly. Lastly, we briefly discuss ways in which we should **test** the code that we write.

9.1 Protecting and Simplifying an Object

When creating and defining an object it is a good idea to keep it simple so that anybody who uses that object in the future (including yourself) can remember how to use it. Often, there are details about an object that we don't need to know about in order to use the object. For example, when we drive a car, we need to know simple things such as:

- starting/ stopping
- steering
- changing gears
- braking, etc..

However, we do not need to worry about things such as:

- assembling the carburetor
- adjusting the spark plug timing
- installing gas lines
- changing the muffler, etc..

Cars are clearly designed to be easy to drive, requiring a simple and easy-to-understand interface. Similarly, it is important that we make our code easy to use and easy to understand. Otherwise, making changes to the code, debugging it and extending it with new features can quickly become very difficult and time consuming.

In order to keep our code simple, we need to make the interface (or "outside view") of our objects as simple as possible. That means, we need to "**hide the details**" of our object that most people would not need to worry about. That is, we need to hide some of the attributes (complicated parts) and methods (complicated procedures) for our object "under the hood", so to speak.

In addition to simplicity, there is another reason to hide some of the details of our object. We would like to prevent outsiders from "messing around with" the inner details of an object. For example we lock our car doors and trunk so that people don't get in there are take things away or damage them etc.. Similarly, for example, if we allow anyone to access the attributes of our object and perform behaviors on it in the wrong order, then this could lead to corrupt data and/or various types of errors in our code.

The idea of hiding the "unnecessary details" of an object from general users is called ***encapsulation*** in JAVA. Encapsulation involves enclosing our object with a kind of "protective bubble" so that it cannot be accessed or modified without proper permission.

In JAVA, we protect and hide attributes and behaviors by using something called an ***access modifier***. This is a big word for something quite simple. Basically, it allows us to set *permissions* for our attributes and methods so that they will be visible/modifiable/usable) from some places in our code but not from other places. As a result, when working with a team of software developers on a large program, some developers will have the freedom to access or modify attributes or methods from various objects, while others will not be allowed such freedom to view or change portions the objects as they would like to.

Protecting Behaviors

We have already been using an access modifier called **public** when we wrote our **public main()** method and **toString()** methods:

```
public static void main(String args[]) { … }
public String toString() { … }
```

The keyword **public** at the front of a method declaration means that the method is publicly available to everyone, so that these methods may be called from anywhere. For all of our other methods however, we did not write **public**, and so they had what is known as **default access** … meaning that the methods may be called from any code that is in the same package or folder that this method's class is saved into. If we write all of our code in the same folder, then **default** and **public** access means the same thing.

There are two other access modifier options available called **private** and **protected**. When we declare a method as **private**, we would not be able to use this method from any class other than the class in which it is defined. **Protected** methods are methods that may be called from the method's own class or from one of its subclasses. So here is a summary of the access modifiers for methods:

• none	- can be called from any class in the same folder
• `public`	- can be called from anywhere
• `private`	- can only be called from this class
• `protected`	- can be called from this class or any subclasses

In this course, most of the methods that we write are **public** methods which allows the most freedom to access and modify our objects. Usually, **private** methods are known as ***helper methods*** since they are often created for the purpose of helping to reduce the size of a larger **public** method or when a piece of code is shared by several methods.

For example, consider bringing in your car for repair. The publicly available method would be called to **repair()** the car. However, many smaller sub-routines are performed as part of the repair process (e.g., **runDiagnostics()**, **disassembleEngine()** etc...). From the point of view of the user of the class, there is no need to understand the inner workings of the repair process. The user of the class may simply need to know that the car can be repaired, regardless of how it is done.

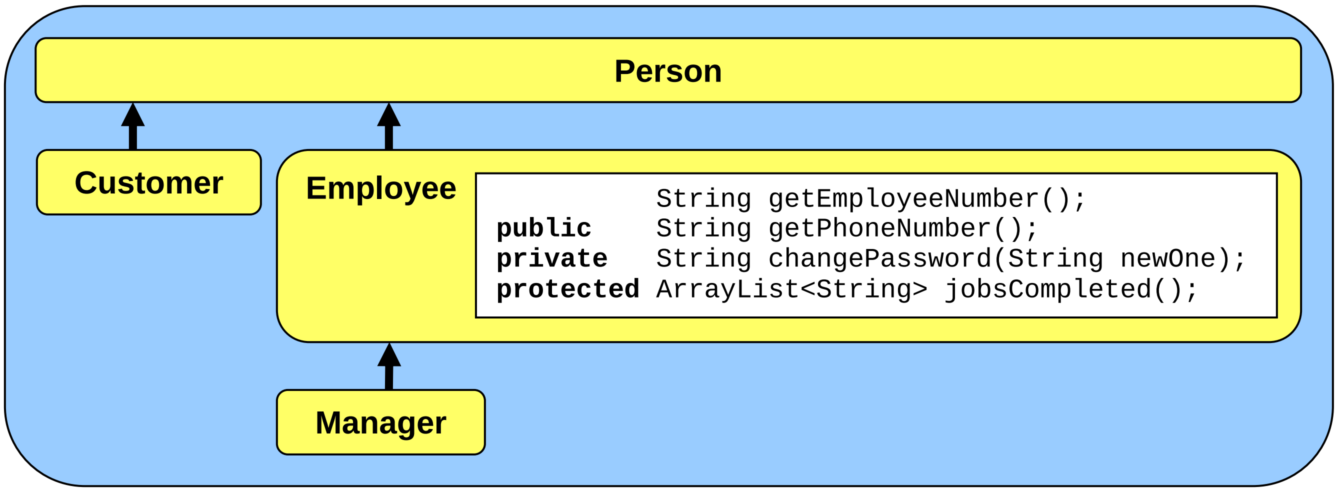
Here is an example of breaking up the repair problem into *helper* methods that do the sub-routines as part of the repair …

```
class Car {
    public void repair() {
        this.runDiagnostics();
        this.disassembleEngine();
        this.repairBrokenParts();
        this.reassembleEngine();
        this.runDiagnostics();
    }
    private void runDiagnostics() { ... }
    private void disassembleEngine() { ... }
    private void repairBrokenParts() { ... }
    private void reassembleEngine() { ... }
}
```

Notice that the helper methods are **private** since users of this class probably do not need to call them. Here is an example showing how we might *attempt* to call these methods from some other class:

```
class SomeApplicationProgram {
    public static void main(String args[]) {
        Car  c = new Car();
        c.repair();                // OK to call this method
        c.disassenbleEngine();     // Won't compile, since it is private
        c.repairBrokenParts();     // Won't compile, since it is private
    }
}
```

Now, to understand the **protected** modifier, we need to consider a class hierarchy. Recall the **Person/Employee/Manager/Customer** example. Consider four methods within the **Employee** class with various access modifiers as follows:



```
          String getEmployeeNumber();
public    String getPhoneNumber();
private   String changePassword(String newOne);
protected ArrayList<String> jobsCompleted();
```

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

```
class Manager extends Employee  {
   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:

```
class Customer extends Person  {
   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 really is not much more to the access modifiers when it comes to methods. However, 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.

Protecting Attributes

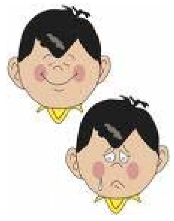
Now what about protecting an object’s attributes ? Well, the **public/private/protected** and *default* modifiers all work the same way as with behaviors. When used on instance variables, it allows others to be able to access/modify them according to the specified restrictions.

So far, we have never specified any modifiers for our attributes, allowing them all *default* access from classes within the same package or folder.

However, in real world situations, it is often best NOT to allow outside users to modify the internal private parts of your object. The reason is that results can often be disastrous. It is easy to relate to this because we well understand how we hide our own private parts ☺.

As an example, consider the following code, which may appear in any class. It shows that we can directly access the **balance** of a **BankAccount**. This is clearly undesirable since there is little protection. Could you imagine if anyone could modify the balance of your bank account directly ?

```
BankAccount myAccount = new BankAccount("Mine");

myAccount.balance = 1000000.00f;  // YAY :)

myAccount.balance = -1000000.00f; // WHY :(
```



In order to prevent direct access to important information we would need to prevent the code above from compiling/running. If we were to declare the **balance** instance variable as **private** within the **BankAccount** class, then the above code would not compile, thus solving the issue.

In general, while freedom to access/modify anything from anywhere seems like a friendly thing to do, it is certainly dangerous. Anyone could "stomp" all over our instance variables changing them at will. A general "rule-of-thumb" that should be followed is to declare ALL of your instance variables as **private** as follows:

```
class Person {
    private String   name;
    private int      age;
    private float    height;
    private char     gender;
    private boolean  retired;

    ...
}
```

Once we do this, then the following code will not work (when written in a class other than the **Person** class):

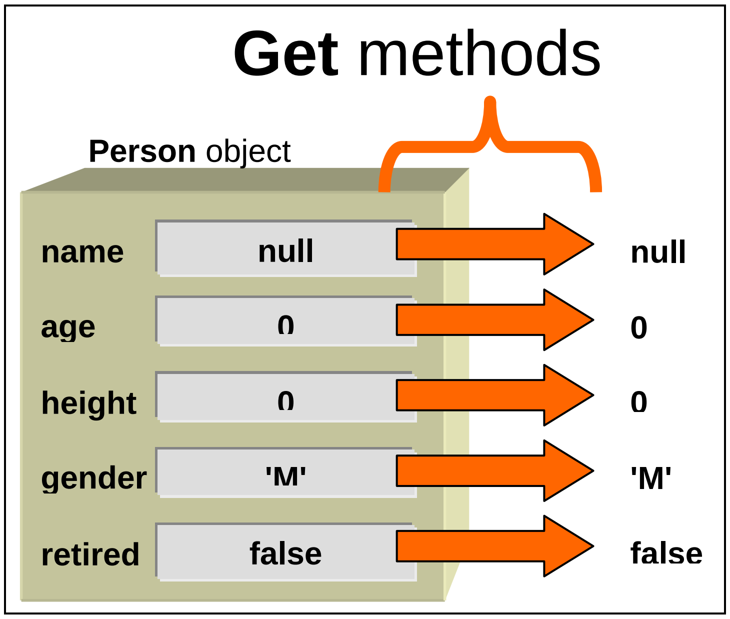
```
class SomeApplicationProgram {
    public static void main(String args[]) {
        Person  p = new Person();
        p.name = "Sandy Beach";    // will NOT compile
        p.age = 15;                // will NOT compile
        p.height = 5.85f;          // will NOT compile
        p.gender = 'M';            // will NOT compile
        p.retired = false;         // will NOT compile
        System.out.println(p.name);     // will NOT compile
        System.out.println(p.age);      // will NOT compile
        System.out.println(p.height);   // will NOT compile
        System.out.println(p.gender);   // will NOT compile
        System.out.println(p.retired);  // will NOT compile
    }
}
```

What we have essentially done is to erect a wall around the object ... like the wall around a city. We have encapsulated it with a protective bubble. Although we are still able to create the object, we are prevented from accessing or modifying its internals now from outside the class. By doing this, we have protected the object so much that we cannot get information neither into it nor out from it. We have kind of secluded the object from the rest of the world by doing this. However, just as a walled city has gates or doors to allow access, we too have a form of gated access by means of any publicly available methods.

We will grant access to "*some*" of our object's attributes (i.e., instance variables) by creating methods known as ***get*** and ***set*** methods (also called ***getters*** and ***setters***). The idea of creating these gateways to our object's data is common practice and is considered to be a robust strategy when creating classes to be used in a large software application. In this course, since we are only creating a few classes and since we are the only code writers, we may not immediately see the benefits of declaring **private** attributes and then creating these methods. However, in a larger/complicated system with hundreds of classes, the benefits become quite clear:

- object attributes are easier to understand and use
- attributes are protected from external/unknown changes
- we are following proper and robust coding style

Let us first consider ***get*** methods. They let us look at information that is within the object by getting the object's attribute values (i.e., get the values of the instance variables):



Get methods have:

- **public** access
- name matching attribute's name
- return type matching attribute's type
- code returning attribute's value

Here is how we would write the standard ***get*** methods for the **Person** class:

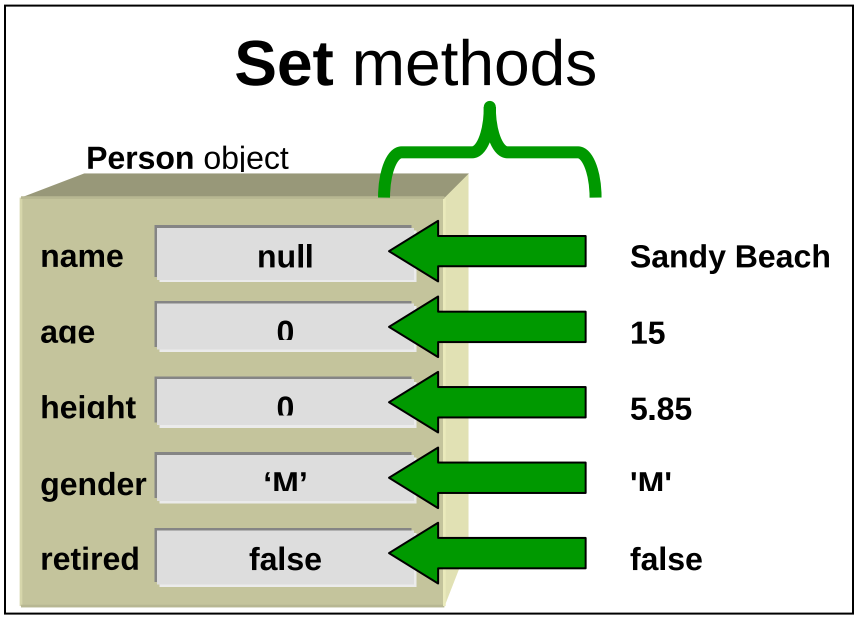
```
public class Person {
    private String   name;
    private int      age;
    private float    height;
    private char     gender;
    private boolean  retired;

    // Get method for name attribute
    public String getName() {
        return this.name;
    }
    // Get method for age attribute
    public int getAge() {
        return this.age;
    }
    // Get method for height attribute
    public float getHeight() {
        return this.height;
    }
    // Get method for gender attribute
    public char getGender() {
        return this.gender;
    }
    // Get method for retired attribute
    public boolean isRetired() {
        return this.retired;
    }
}
```

Notice that all the methods look the same in structure. They are all **public**, all have return types and names that match the attribute type, all have no parameters and all are one line long. When we call the method to get the attribute value, the method simply returns the attribute value to us. Its quite simple. By convention, all get methods start with "**get**" followed by the attribute name, with the exception of attributes that are of type **boolean**. In that case, we usually use "**is**" followed by the attribute name, as it makes the method call more natural.

Now let us examine the ***set*** methods. ***Set*** methods allow us to put values into the instance variables (i.e., to set the object's attributes):

Set methods have:

- **public** access
- **void** return type
- name matching attribute's name
- a parameter matching attribute's type
- code giving the attribute a value



Here is how we would write the standard ***set*** methods for the **Person** class:

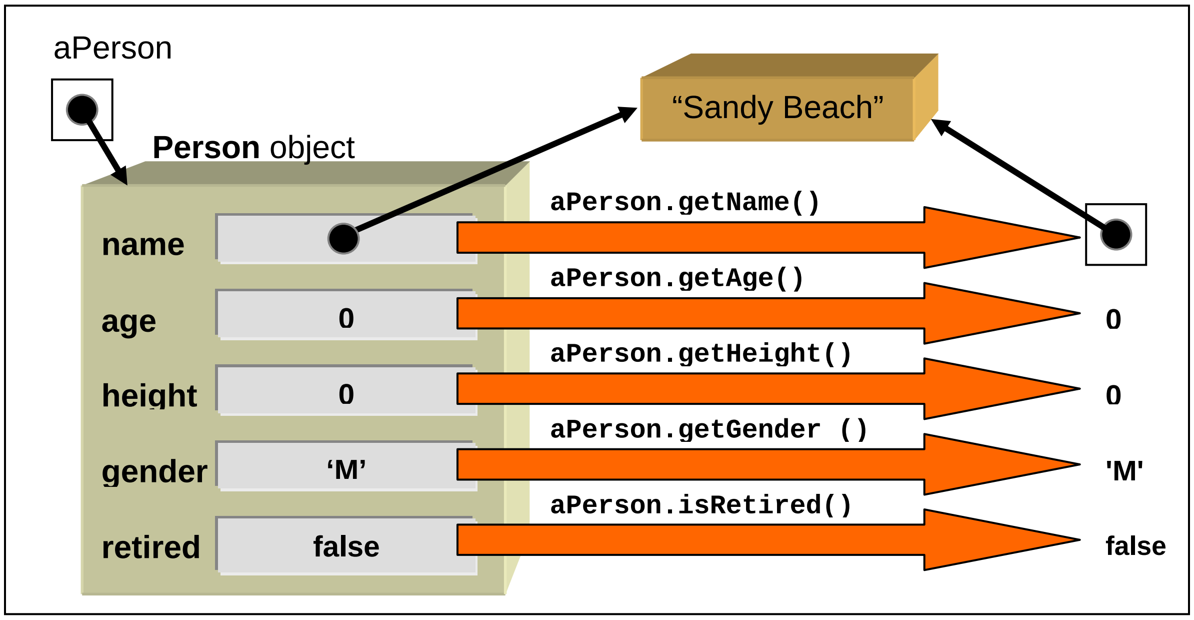
```
// Set method for name attribute
public void setName(String n) {
    this.name = n;
}
// Set method for age attribute
public void setAge(int a) {
    this.age = a;
}
// Set method for height attribute
public void setHeight(float h) {
    this.height = h;
}
// Set method for gender attribute
public void setGender(char g) {
    this.gender = g;
}
// Set method for retired attribute
public void setRetired(boolean r) {
    this.retired = r;
}
```

The single line of code in a ***set*** method is quite simple also.
When we call the method to give the attribute a new value (i.e., we supply the new value as a parameter to the method), the method simply takes that new attribute value and sets the attribute to it by using the **=** operator.

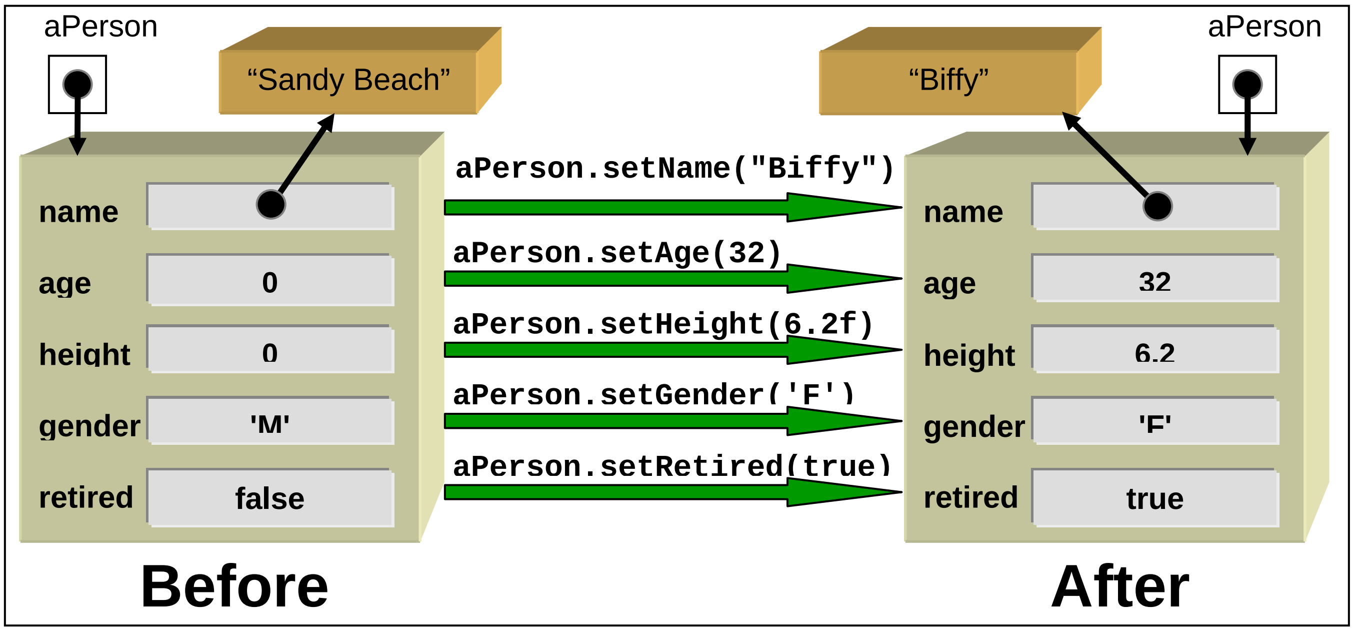
Normally, we write all the ***get*** and ***set*** methods together, and sometimes shorten them onto one line. Also, they are often listed in the code right after the **public** constructors as follows:

```
class Person {
    private String   name;
    private int      age;
    private float    height;
    private char     gender;
    private boolean  retired;

    // Constructors
    public Person() { /*...*/ }
    public Person(String n, int a, float h, char g, boolean r) { /*...*/ }

    // Get methods
    public String getName() { return this.name; }
    public int getAge() { return this.age; }
    public float getHeight() { return this.height; }
    public char getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }

    // Set methods
    public void setName(String n) { this.name = n; }
    public void setAge(int a) { this.age = a; }
    public void setHeight(float h) { this.height = h; }
    public void setGender(char g) { this.gender = g; }
    public void setRetired(boolean r) { this.retired = r; }
}
```

Here is how the ***get*** method works:



Notice that primitive attribute values are returned as simple values but *object* attribute values are returned as pointers to the object. The **Person** object remains unchanged as a result of a get method call. Here is how the ***set*** method works:



Notice that primitive attribute values are simply replaced with the new value. For object attribute values, after the set call, the attribute will point to the new object. The previous object that the attribute used to point to is discarded (i.e., garbage collected) if no other objects are holding on to it. Once we create these ***get/set*** methods, we can then access and modify the object from anywhere in our program as before:

```
class TestPersonProgram {

    public static void main(String args[]) {
        Person  p = new Person();

        System.out.println("Before Setting ...");
        System.out.println(p.getName());    // was println(p.name);
        System.out.println(p.getAge());     // was println(p.age);
        System.out.println(p.getHeight());  // was println(p.height);
        System.out.println(p.getGender());  // was println(p.gender);
        System.out.println(p.isRetired());  // was println(p.retired);

        p.setName("Sandy Beach");           // was p.name = "Sandy Beach";
        p.setAge(15);                       // was p.age = 15;
        p.setHeight(5.85f);                 // was p.height = 5.85f;
        p.setGender('F');                   // was p.gender = 'F';
        p.setRetired(true);                 // was p.retired = true;

        System.out.println("\nAfter Setting ...");
        System.out.println(p.getName());    // was println(p.name);
        System.out.println(p.getAge());     // was println(p.age);
        System.out.println(p.getHeight());  // was println(p.height);
        System.out.println(p.getGender());  // was println(p.gender);
        System.out.println(p.isRetired());  // was println(p.retired);
    }
}
```

Here is what the output would be (however, initial values depend on the **Person** constructor):

```
Before Setting ...
"UNKNOWN"
0
0.0
?
false

After Setting ...
"Sandy Beach"
15
5.85
F
true
```

Now if we think for a moment ... what did we really do by making all the ***get*** and ***set*** methods ? Really, we wrote a lot of code (e.g., 5 ***get*** methods and 5 ***set*** methods for the **Person** class) but did not gain anything new. The code does the same thing as before. In fact, the test code seems longer and perhaps slower (since we are calling a method to get/set the instance variables for us instead of accessing them directly). So why did we do this ? Lets review the advantages again:

1. First, ***get/set*** methods actually make life simpler for users of your class because the user does not have to understand the "guts" of the object being used. It allows them to treat the object as a "black box". The user does not need to know about all the instance variables. Some are used to hold data that is temporary or private. You should only create public ***get*** methods for the instance variables that the user of the class would need to know about.

2. Second, it prevents the users of a class from directly modifying the object's internals. Recall, for example, that we should never be able to directly change the balance of our bank account without going through the proper transaction procedures such as depositing and withdrawal. Of course, if we always create **public *get/set*** methods for all our attributes, then we still would have no such protection. So, it is important to create ***set*** methods **only** for the attributes that you want the user of the class to be able to change directly. Therefore, you do not always need to make ***set*** methods.

Protecting Classes

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 been default access, but we can write **public** in front of the class if we want this to be a truly publicly accessible object:

```
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 ☹.

9.2 Simplifying Constructors and Eliminating "this"

You may have noticed that we have a lot of duplicated code in our 3 **Person** constructors that we wrote earlier. Each attribute of the object was set explicitly with a line of code like this:

```
this.<attribute> = <initialValue>;
```

Here was the code …

```
// This is the zero-parameter constructor
Person() {
      this.firstName = "UNKNOWN";
      this.lastName = "UNKNOWN";
      this.gender = '?';
      this.retired = false;
      this.age = 0;
      this.address = null;
}

// This is a 4-parameter constructor
Person(String fn, String ln, char g, boolean r) {
      this.firstName = fn;
      this.lastName = ln;
      this.gender = g;
      this.retired = r;
      this.age = 0;
      this.address = null;
}

// This is a 6-parameter constructor
Person(String fn, String ln, int a, char g, boolean r, Address adr) {
      this.firstName = fn;
      this.lastName = ln;
      this.age = a;
      this.gender = g;
      this.retired = r;
      this.address = adr;
}
```

We can actually reduce the amount of duplicated code by ***chaining*** our constructors together (i.e., calling one constructor from another). We do this by making use of the **this** keyword again. Notice the reduced code:

```
// This constructor calls the 6-parameter one
Person() {
      this("UNKNOWN", "UNKNOWN", 0, '?', false, null);
}

// This constructor calls the 6-parameter one
Person(String fn, String ln, char g, boolean r) {
      this(fn, ln, 0, g, r, null);
}

// This is the 6-parameter constructor
Person(String fn, String ln, int a, char g, boolean r, Address adr) {
      this.firstName = fn;
      this.lastName = ln;
      this.age = a;
      this.gender = g;
      this.retired = r;
      this.address = adr;
}
```

Notice that the first 2 constructors simply call the last one which takes all 6 parameters as arguments. You may also notice that there is no dot **.** operator after the **this** keyword. This is a special use of the keyword **this** which only applies when calling one constructor from another.

Notice as well in the above code that the keyword **this** is also used to access the object's attributes. As it turns out, whenever we are writing a constructor or instance method of a class, we do not need to use the keyword **this** to access the attributes of the object or not call another instance method in the same class. We can leave off **this** completely, and JAVA will assume that we meant "**this**" object. Hence, the following constructor is equivalent to the 3rd one above:

```
Person(String fn, String ln, int a, char g, boolean r, Address adr) {
      firstName = fn;        // same as   this.firstName = fn;
      lastName = ln;         // same as   this.lastName = ln;
      age = a;               // same as   this.age = a;
      gender = g;            // same as   this.gender = g;
      retired = r;           // same as   this.retired = r;
      address = adr;         // same as   this.address = adr;
}
```

Note however, that we cannot remove the keyword **this** in the chained constructors that use **this** without the dot afterwards, so the first two constructors must remain the same.

In fact, we can go through our entire code and remove the code **this.** from our instance methods since all instance methods have direct access to their own internal attributes. Here are some methods that we wrote in the **Person** class. Notice how we can remove all occurrences of **this.** from the code:

```
int computeDiscount() {
    if ((~~this.~~gender == 'F') && (~~this.~~age < 13 || ~~this.~~retired))
        return 50;
    else
        return 0;
}

boolean isOlderThan(Person  x) {
    return (~~this.~~age > x.age);
}

boolean isOlderThan(Person  x, Person  y) {
    if ((~~this.~~age > x.age) && (~~this.~~age > y.age))
        return true;
    else
        return false;
}
```

```
Person oldest(Person  x, Person  y) {
    if ((~~this.~~age > x.age) && (~~this.~~age > y.age))
        return this;  // cannot remove this here though
    else if ((x.age > ~~this.~~age) && (x.age > y.age))
        return x;
    else
        return y;
}

void swapNameWith(Person x) {
    String      tempName;

    tempName = ~~this.~~firstName;
    ~~this.~~firstName = x.firstName;
    x.firstName = tempName;

    tempName = ~~this.~~lastName;
    ~~this.~~lastName = x.lastName;
    x.lastName = tempName;
}

public String toString() {
    String answer = ~~this.~~age + " year old ";

    if (!~~this.~~retired)
       answer = answer + "non-";

    return ("retired person named " +
            ~~this.~~firstName + " " + ~~this.~~lastName);
}
```

Recall as well in the **Team/League** example that we called one instance method from within another. Here too, for each call that we made within the same class, we can remove the references to **this.** since JAVA assumes automatically that we meant "this" class if we leave it out. Here are some examples from the **Team/League** example showing where it can be removed:

```
void recordWinAndLoss(String winnerName, String loserName) {
    Team   winner, loser;

    winner = ~~this.~~teamWithName(winnerName);
    loser = ~~this.~~teamWithName(loserName);
    ~~this.~~recordWinAndLoss(winner, loser);
}

void recordTie(String teamAName, String teamBName) {
    Team   teamA, teamB;

    teamA = ~~this.~~teamWithName(teamAName);
    teamB = ~~this.~~teamWithName(teamBName);
    ~~this.~~recordTie(teamA, teamB);
}
```

As a rule of thumb, you can ALWAYS remove **“this.”** whenever it appears in your code since JAVA will automatically assume that you are trying to access attributes for (or call a method for) the receiver object when no object is specified. Until now, we have been using **this.** throughout our code because it makes it easier to understand which object we are dealing with. However, it is common practice to leave **this.** out of our code to reduce the clutter.

9.3 Type-Casting, Polymorphism and Double-Dispatching

Recall that we can 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
```

Remember that some type-casting is done automatically by JAVA whereas in other cases we can 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.

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)anArrayList.get(i);
b = (SavingsAccount)aBankAccount;
```

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

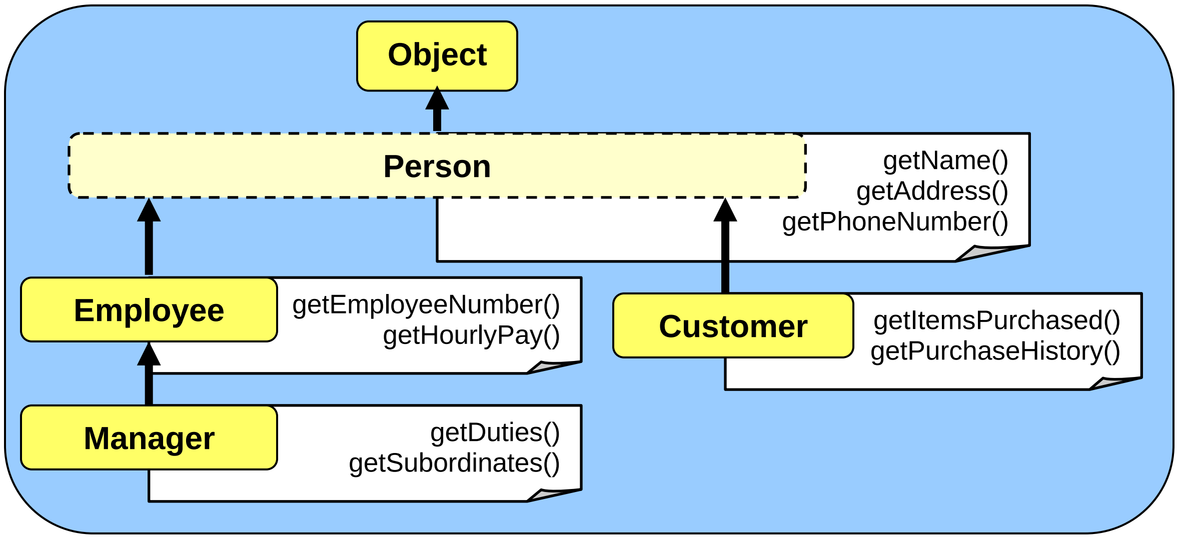
When we type-cast an object to another type we are not modifying it in any way. Rather, we are simply causing the object to be **“treated”** more generally from then on in the program. As a result, the object will then be less flexible in that we can no longer call some of the methods that we used to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

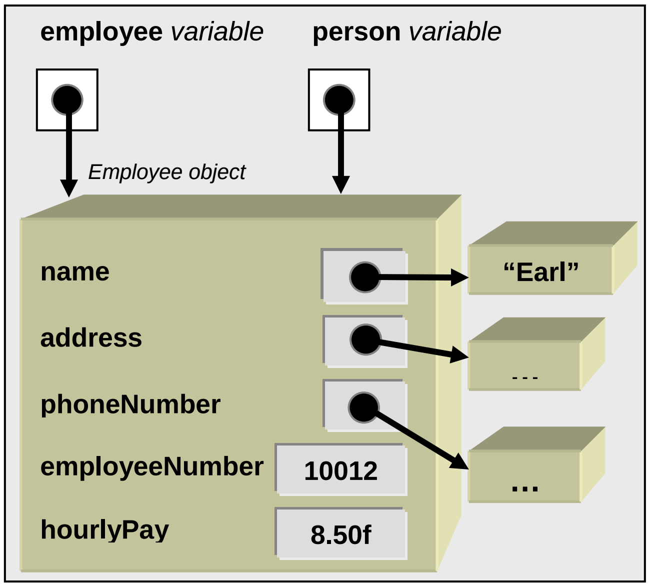
As another example, consider an **Apple** … normally you may **polish**, **peel** or **eat** it ... but in a food fight, you may type-cast (i.e., treat) your apple as a general ***throwable*** projectile. Then, the apple takes on different behavior such as **throw**, **catch**, **splatter**, etc... The 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.. a dangerous food fight.

Now lets 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.

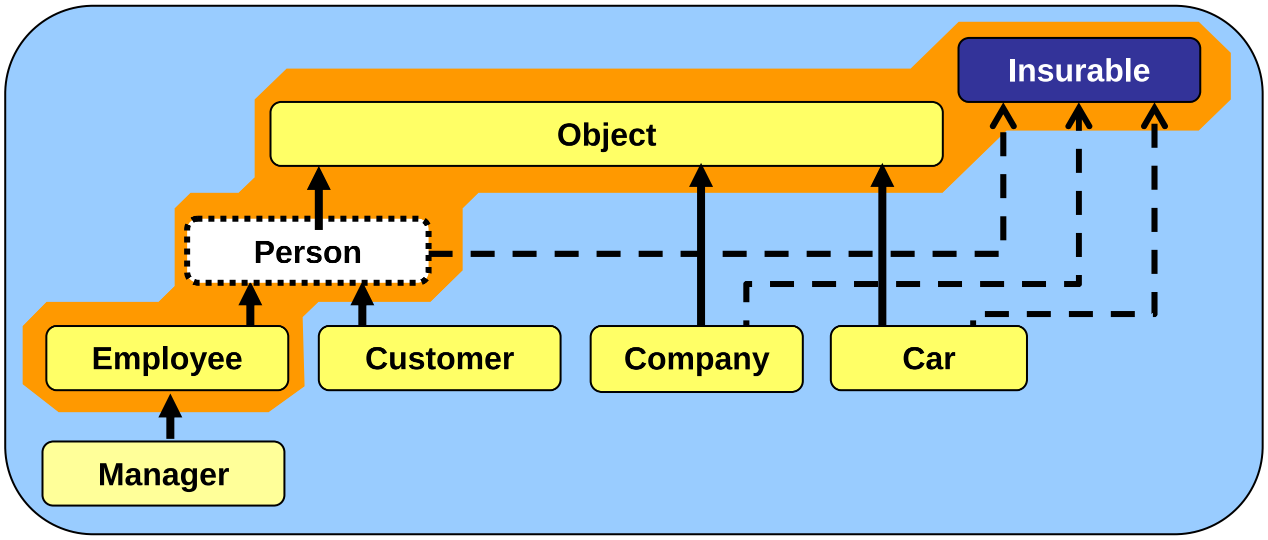
Notice what we are not able to do:

```
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.**

Why would we want to do type-casting in the first place ? It seems that all 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*** which 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" messages (i.e., 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...
- and, we can deposit to any **BankAccount**, independent of its type.
- all **Objects** understand the **toString()** method, so we do not have to remember additional method names.

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.

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;
```

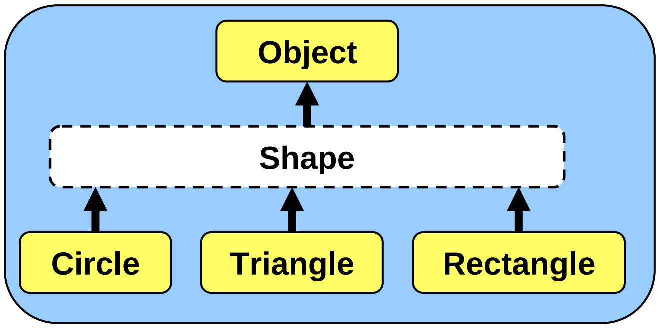
2. when we pass in the object as a parameter to a method which has a more general type:

```
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 Rectange(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 ()**:

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

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

```
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 **aShape** variable. How then do we know which **draw()** method to call ? Well, we could check the type of the object, perhaps with the **instanceof** keyword 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);
```

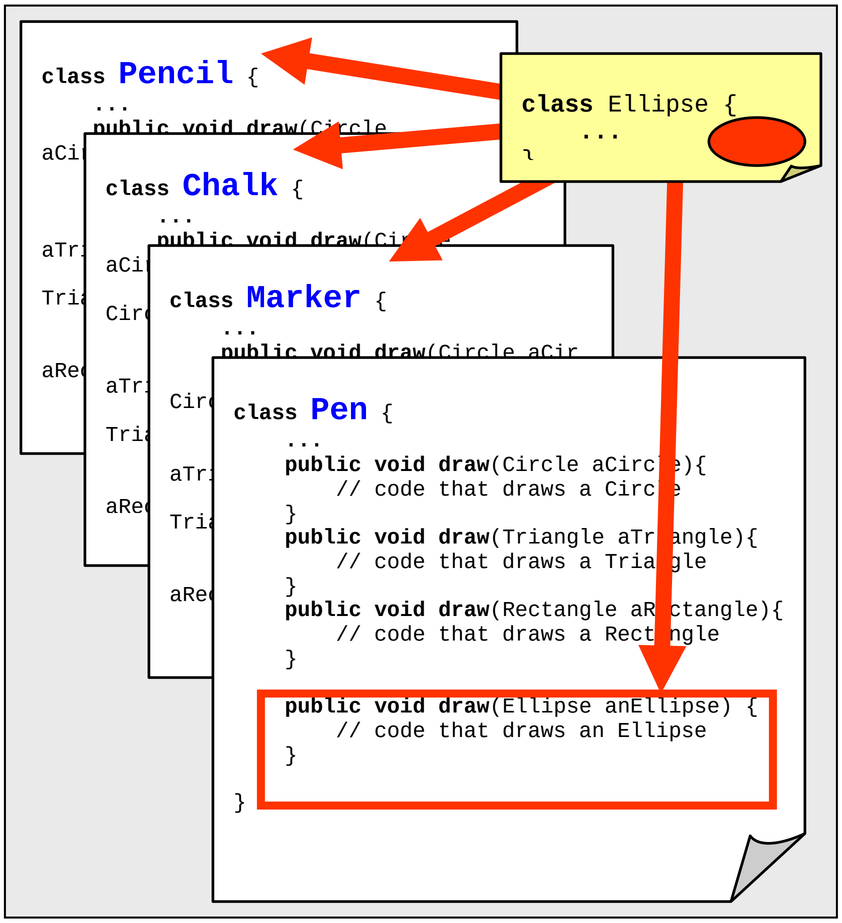
However, this is not so straight forward. We would have to define a **draw()** method in the **Pen** class for each kind of shape in order to satisfy the compiler with regards to the particular type of the parameter:

```
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.



```
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
    }

    public void draw(Ellipse anEllipse) {
        // code that draws an Ellipse
    }
}
```

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 …

```
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:

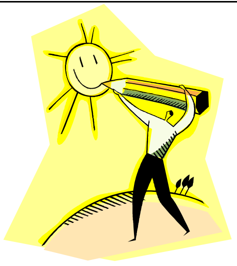
```
class Circle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

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

```
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:

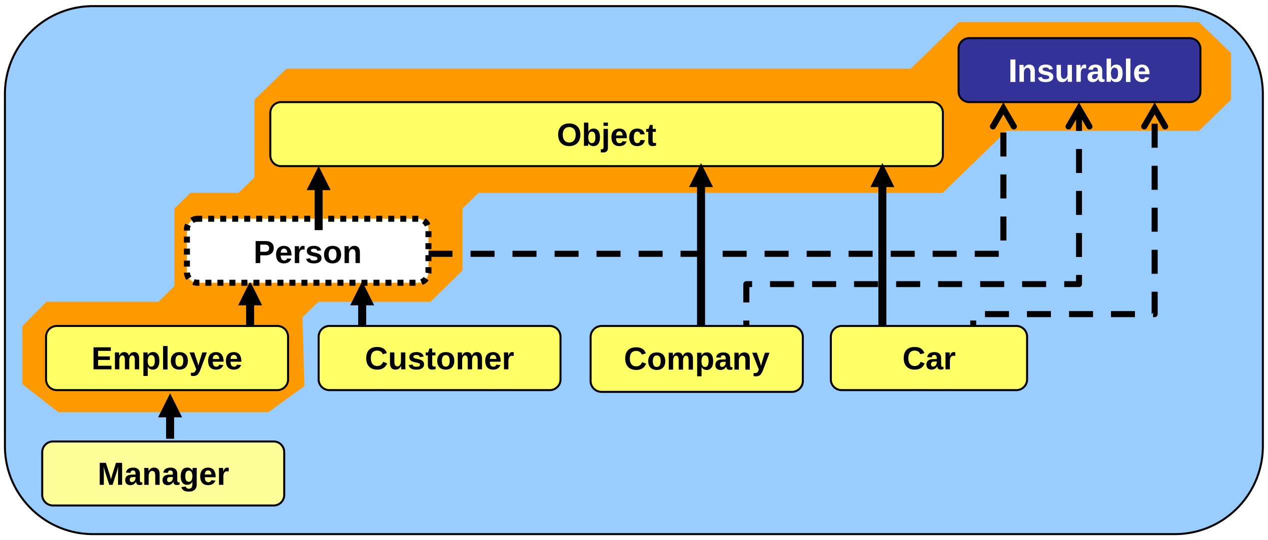
```
class Pen {
    ...
    public void draw(Shape aShape) {
         aShape.drawWith(this);
    }
}
```



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 adding a variety of **Insurable** items to an **ArrayList** and then listing all of the policies and totaling the amounts of all the policies:

```
float                     total = 0;
ArrayList<Insurable>  insurableItems;

insurableItems = new ArrayList<Insurable>();
insurableItems.add(new Car("Porshce", "Carerra", "Red", 340));
insurableItems.add(new Customer("Guy Rich"));
insurableItems.add(new Company("Elmo’s Edibles", 2009));
insurableItems.add(new Employee("Jim Socks"));
insurableItems.add(new Manager("Tim Burr"));

System.out.println("Here are the policies:");
for (Insurable item: insurableItems) {
    System.out.println("  " + item.getPolicyNumber());
    total += item.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 **ArrayList**. 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 **ArrayList** 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 **ArrayList<Object>** of general objects.

```
ArrayList<Object>     insurableItems = new ArrayList<Object>();
```

This will affect the ***type*** defined for our FOR loop as well:

```
for (Object item: insurableItems) { ... }
```

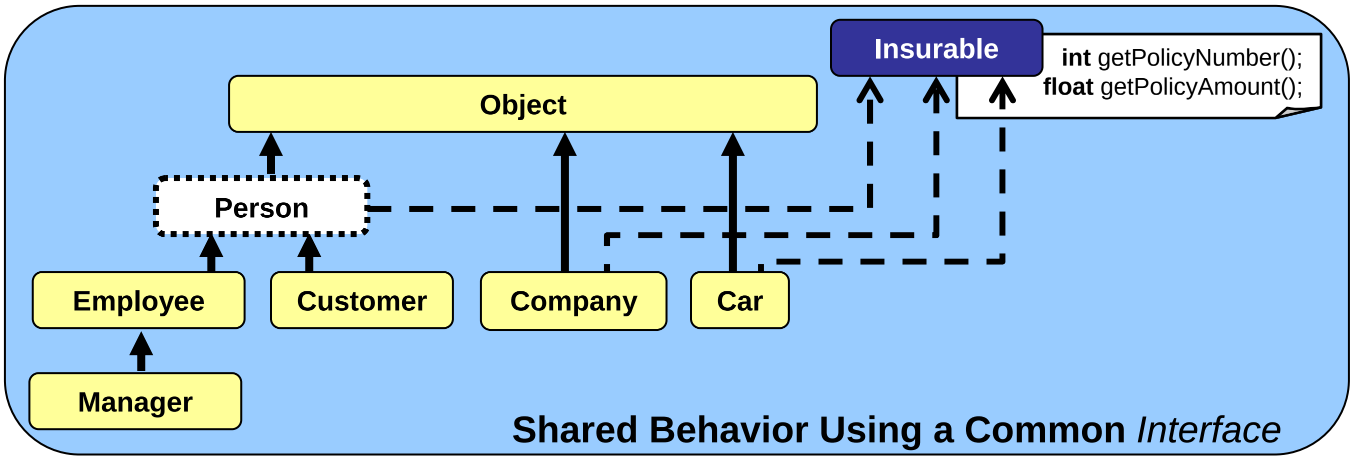
Once we make these changes, then the compiler will prevent us from calling the **getPolicyNumber()** or **getPolicyAmount()** methods because it assumes that the **item** extracted in the FOR loop is a general **Object** … but general objects do not have such methods. Therefore, we would be forced to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

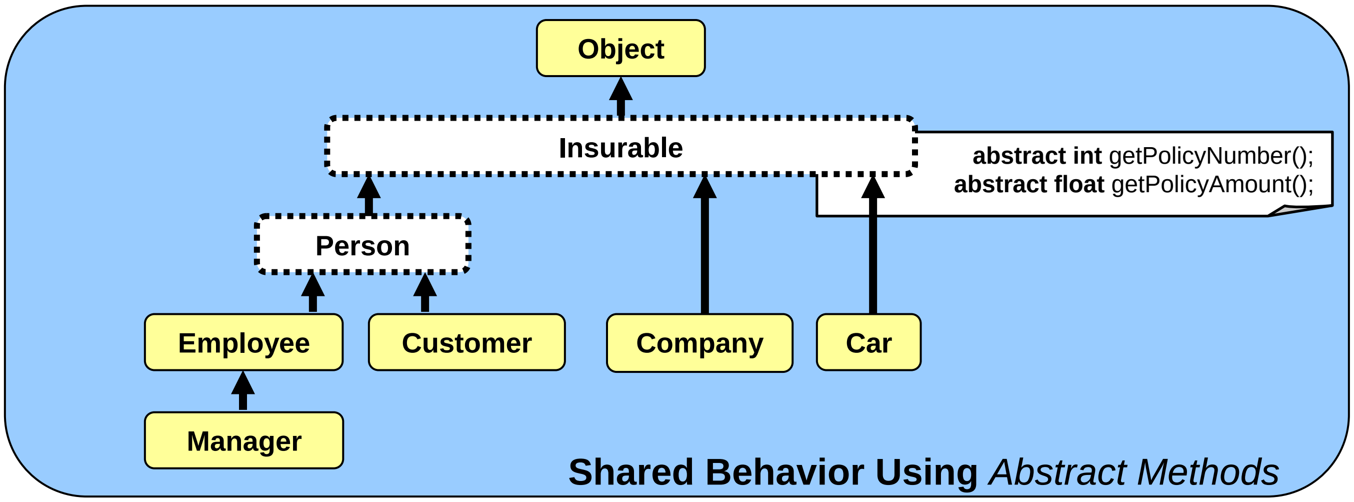
```
...
for (Object item: insurableItems) {
    if (item instanceof Car)
        System.out.println("  " + ((Car)item).getPolicyNumber());
        total += ((Car)item).getPolicyAmount();
    }
    else if (item instanceof Employee)
        System.out.println("  " + ((Employee)item).getPolicyNumber());
        total += ((Employee)item).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.

The big disadvantage though 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 …



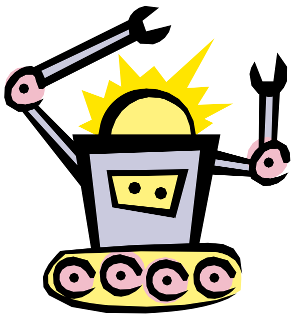

Object
Insurable
abstract int getPolicyNumber();
abstract float getPolicyAmount();
Person
Employee
Customer
Company
Car
Manager
Shared Behavior Using *Abstract Methods*

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 ArrayList<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 **RemoteContrrol** class code whatsoever:

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

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

9.4 Proper Testing

As you know now, object-oriented programming requires you to define and implement many objects and to get them to work together in meaningful ways. Often, the object class definitions that you write can be re-used in many applications. It is therefore a good idea to ensure that these objects are robust and that their methods provide proper results. To do this, you should perform proper testing of your objects.

Unfortunately, testing is often tedious. It is therefore poorly done and ignored by many programmers. Companies that hire programmers do not like laziness … and even worse … they hate code with bugs or errors in it. To avoid disappointing your boss, possibly losing your job, and just to feel good about the quality of your work … you should properly test your code.

Normally, it is not common to test your constructors nor get/set methods, but it is certainly important to test methods that perform computations, search, sort, etc… For problems that require numerical parameters, it is a good idea to test different values that could potentially cause problems. For example, if we were to fully test the **deposit()** method for the **BankAccount** class, we would want to test depositing the following amounts:

```
• 0.0           // deposit nothing
• 0.67          // a cents amount
• 100.57        // a typical positive amount
• 100.2234343   // an amount with many decimal places
• -34           // an invalid amount
```

We could create a simple test program to do this, making sure that we properly display the results to confirm that they are correct as follows …

```java
class BankAccountTestProgram {
    public static void main(String args[]) {
        BankAccount   acc;

        acc = new BankAccount("Rusty Can");
        System.out.println("Account at start:                      " + acc);
        acc.deposit(0.0f);
        System.out.println("Account after depositing $0.00:        " + acc);
        acc.deposit(0.67f);
        System.out.println("Account after depositing $0.67:        " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57:      " + acc);
        acc.deposit(100.2234343f);
        System.out.println("Account after depositing $100.2234343:" + acc);
        acc.deposit(-34);
        System.out.println("Account after depositing $-34:         " + acc);
    }
}
```

Here is the output:

```
Account at start:                      Account #100000 with $0.0
Account after depositing $0.00:        Account #100000 with $0.0
Account after depositing $0.67:        Account #100000 with $0.67
Account after depositing $100.57:      Account #100000 with $101.24
Account after depositing $100.2234343:Account #100000 with $201.46344
Account after depositing $-34:         Account #100000 with $167.46344
```

Notice the careful use of **System.out.println()** in the program to provide a kind of “log” showing exactly what we tested and the order that things were tested in. If you were to read the output, you should be able to follow along as the deposit transactions were made to confirm the correct balance each time.

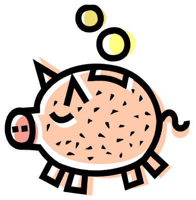
From the output, you may notice a few things that you would like to change. First, you may feel that the **toString()** method would be better if it displayed the money amounts properly with 2 decimal places. You can then change your code accordingly and re-run the test:

```
Account at start:                      Account #100000 with $0.00
Account after depositing $0.00:        Account #100000 with $0.00
Account after depositing $0.67:        Account #100000 with $0.67
Account after depositing $100.57:      Account #100000 with $101.24
Account after depositing $100.2234343:Account #100000 with $201.46
Account after depositing $-34:         Account #100000 with $167.46
```

Second, you may decide to disallow depositing negative amounts of money. You might do this by changing the code to generate an exception or perhaps simply perform a check and ignore deposits of negative amounts.

It really depends on the application and whether or not it is tied-in with the user interface. For example, at a bank machine, it is impossible to deposit a negative amount of money because the machine does not allow you to enter a negative sign. In such a situation, you may choose simply to ignore the problem altogether, since it would never occur. However, a simple check may be best, in case you port your code into a different program:

```
void deposit(float amount) {
    if (amount > 0)
        balance += amount;
}
```



Then we would re-run the same test code to see whether or not it worked:

```
Account at start:                  Account #100000 with $0.00
Account after depositing $0.00:    Account #100000 with $0.00
Account after depositing $0.67:    Account #100000 with $0.67
Account after depositing $100.57:  Account #100000 with $101.24
Account after depositing $100.2234343:Account #100000 with $201.46
Account after depositing $-34:     Account #100000 with $201.46
```

Now this was a simple test program which is often known as a “Test Unit”. In larger, more complicated, real-word programs, in order to keep organized, it would be necessary to create multiple simple test units that test particular aspects of the program. For example,

```
class BankAccountTestUnit1 {
    public static void main(String args[]) {
        BankAccount   acc = new BankAccount("Rusty Can");

        System.out.println("Account before depositing $100.57:   " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57:    " + acc);
    }
}
```

```
class BankAccountTestUnit2 {
    public static void main(String args[]) {
        BankAccount   acc = new BankAccount("Rusty Can");

        System.out.println("Account before withdrawing $100.57:   " + acc);
        acc.withdraw(100.57f);
        System.out.println("Account after withdrawing $100.57:    " + acc);
    }
}
```

In fact, it is often the case that we would like to perform transactions and test cases on a particular bank account. In this case, we can breakdown the separate test units as test methods in a larger test program:

```
class BankAccountTestUnit3 {
    static void deposit1(BankAccount  acc) {
        System.out.println("Account before depositing $100.57:  " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57:   " + acc);
    }
    static void deposit2(BankAccount  acc) {
        System.out.println("Account before depositing $0.01:  " + acc);
        acc.deposit(0.01f);
        System.out.println("Account after depositing $0.01:   " + acc);
    }
    static void withdraw1(BankAccount  acc) {
        System.out.println("Account before withdrawing $100.57:  " + acc);
        acc.withdraw(100.57f);
        System.out.println("Account after withdrawing $100.57:   " + acc);
    }
    static void withdraw2(BankAccount  acc) {
        System.out.println("Account before withdrawing $0.01:  " + acc);
        acc.withdraw(0.01f);
        System.out.println("Account after withdrawing $0.01:   " + acc);
    }

    public static void main(String args[]) {
        BankAccount   acc;

        acc = new BankAccount("Rusty Can");
        deposit1(acc);
        deposit2(acc);
        withdraw1(acc);
        withdraw2(acc);

        acc = new BankAccount("Rusty Door", 10.00f);
        deposit1(acc);
        deposit2(acc);
        withdraw1(acc);
        withdraw2(acc);

        acc = new BankAccount("Rusty Pail", -200.00f);
        deposit1(acc);
        deposit2(acc);
        withdraw1(acc);
        withdraw2(acc);
    }
}
```

There are actually principles and guidelines for writing test cases for large systems. However, it is beyond the scope of this course. You will learn more about proper testing next year.