Encapsulation

In object oriented design, we divide our program into classes representing components of the situation, using instance variables to store all the data we need to know about that component, and methods that store the behaviors and actions related to that component. The different classes can interact, but are separate, meaning that everything we need for one component of the situation is stored in one class. This is called encapsulation.

We have some standard tools that every class with instance variables should have that support clean encapsulation. If a class has instance variables, it must also have methods called accessors, mutators, and constructors (we already know it should have toString).

In addition to helping us design large scale programs by making the structure of the code reflect our conceptual understanding of the situation in the program, object orientation helps programmers be lazier in actually implementing and maintaining code.

Encapsulation means classes can be implemented independent of each other. This allows different programming teams to work in parallel on different classes. It also means that we can pull classes out of one program and copy them into another program without having to re-write from scratch. And it means that when we later have to update or change the code in one class, those changes can be made without having to update all the code in the other classes.

But to get these benefits from object orientation, we have to follow rules for good object oriented design, and this means separating a class' interface from its implementation.

Interface vs Implementation

When creating a class, we should clearly distinguish between the *public interface* of the class, which consists of the parts of the class that the rest of the program can interact with, and the *private implementation* which is the parts of the class that are not available to the rest of the program, the parts that make the interface work.

The public interface for a class should be part of the initial design, and defines the parts of the class that other classes may need to use in their code – this is the way other classes can be written to be ready to interact with this class while the class is still being written. In general, the public interface of a class consists of a list of methods, some of which will return data values for the characteristics (instance variables).

It is important that once a method is put into the public interface, it should not be changed – not the name, not the return type, not the parameters. Also we cannot remove methods

from the interface. By putting something into the public interface, we are promising that it is a tool that other classes can freely use, so changing any of this would break another class' code (which will probably enrage our programming peers!).

Any part of the class that is in the private implementation can be changed as needed to improve the code, fix errors, add new abilities, or meet the demands of management.

We can always go into methods and change their bodies to change how they work, which means we can fix problems, make code more efficient, or improve readability without messing up other classes that call those methods. Method bodies are automatically part of the private implementation just because of how methods work.

The private implementation usually also includes all instance variables, and it may include some methods that are not appropriate for use outside the class.

Hiding instance variables in the private implementation gives each class control of those variables' values, so no class from the outside can set them to inappropriate values. It does also allow us to completely hide the value and even existence of an instance variable that needs to be hidden from the rest of the program for privacy or security reasons.

Java gives us the keywords public and private to explicitly say which parts of the class are in the public interface vs the private implementation.

```
// class Account
public class Account {
    public String name; // public (just for this example)
    private double balance; // private (correct)

    // a public method that controls access
    // to a private variable
    public void deposit(double d) {
        // don't allow negative values
        if (d > 0) {
            balance = balance + d;
        }
    }
}

// main method
public static void main(String[] args) {
    Account act = new Account();
```

```
    // name is currently public, so can be changed from outside
    // that's probably a bad idea
    act.name = "Johann Dowe";
    act.name = "stupid customer I hate him";
    act.name = "";

    // error, cannot access private
    // from outside class
    act.balance = -1;

    // legal, use the public method
    act.deposit(10);
}
```

Accessors and Mutators

Since instance variables are almost always private, it is standard to provide tools that allow outside classes to interact with them in a controlled way. In general, for each instance variable, we would provide a method to access the value of the variable, and a method to try to change the value of the variable.

The method to get access to the value is called an *accessor* or *getter* because the method is usually given a name that puts "get[1]" in front of the name of the variable. Accessors are almost always very simple: just return the value of the variable.

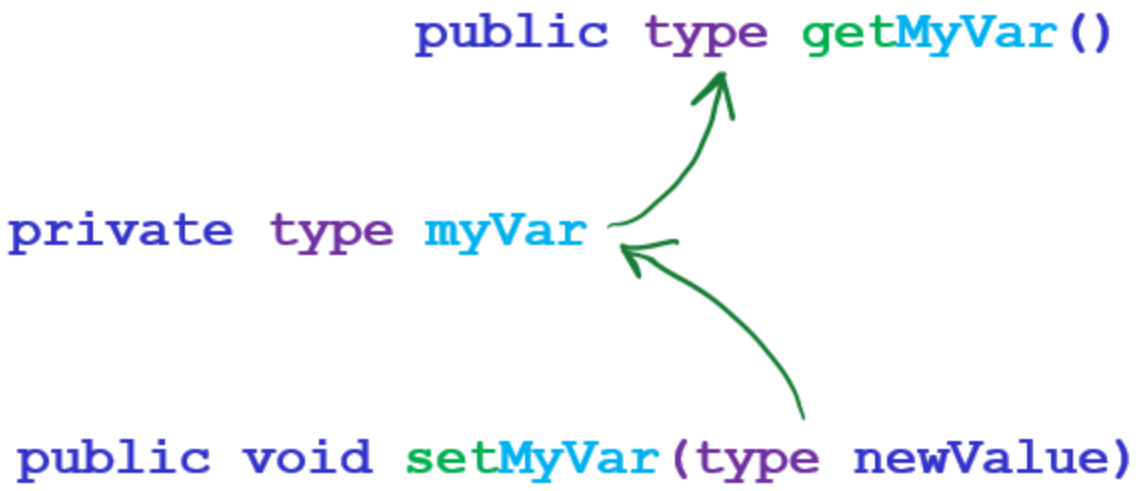
The method to change the value is called a mutator or setter because the method is usually given a name that puts "set" in front of the name of the variable. Mutators need a parameter of the right type for the variable. At simplest they would simply put that value into the variable, but they might also include an if to check the value of the parameter.

```
// class Job
public class Job {
    // monthly salary for this job
    private double salary;
```

[1] An alternative standard for accessors uses `is` in front of the variable name just for boolean variables, so it might be `isHungry()` or `isTall()`. This is a little more common than the standard we are using, and is fine for you to use; to keep things simpler I am sticking with `get` for all types.

```
    // accessor
    public double getSalary() {
        return salary;
    }

    // mutator
    public void setSalary(double newval) {
        if (newval >= 0) {
            salary = newval;
        }
    }
}

public static void main(String[] args) {
    Job myJob = new Job();
    myJob.setSalary(2700);
    System.out.println("you got a 1% raise");

    // new salary is 101% of the old salary
    myJob.setSalary(myJob.getSalary() * 1.01);

    // no effect if we try to set it to a bad value
    myJob.setSalary(-300);

    System.out.println("Your salary is: $" + myJob.getSalary());
}
```

Notice that when putting get and set in front of the instance variable name to create the names of accessors and mutators, we always camel-case the variable name. Since the accessor needs to return the instance variable's value, its return type is always the same name as the type of the variable. Since the mutator's job is to set the value of the instance variable based on the parameter, the parameter type is always the same as the instance variable.



Mutators often need to *validate* the value going into the instance variable, so that it never gets set to an inappropriate value. To do this, they include an if that checks that the value of the parameter (not the instance variable itself) is good, so that the instance variable is only changed if the parameter value is good. However, they do not use an else for bad values[2].

Mutators do not print errors if they are given a bad value! A print-out is a message to the user, but we don't know whether the value passed in to the mutator came directly from the user, it might have come from a file, or been generated by other code in the program. We would say that the mutator is a *back-end* tool, one whose job is to interact with the rest of the program, but not to interact with the user. We absolutely might have other methods that do talk to the user (*front-end*) and make similar checks and tell the user if their input is bad, but then those methods should still use the mutator. The mutator is the last line of defense for the instance variable if a bad value gets through every other part of the code.

Also, **mutators do not set the instance variable back to some other value** if they are given a bad value. Suppose in a Bank class we have an instance variable balance, and the mutator does not allow negative values. If your current balance is 2000.00, and some part of the program tries to set it to -1.00, do you want the mutator to punish the program for the bad value by throwing away the 2000.00 and setting your balance to 0.00? No. If we have a good value currently, we should hold onto that good value until we are asked to change it to another acceptable value.

When given bad values, mutators just silently do nothing.

---

[2] There is an alternative standard for mutators that returns a boolean and returns false if given a bad value, but the standard we are using is far more common.

Accessors and Mutators are another example of a standardized way of writing Java code that Java programmers have agreed on even though it they are not built into the syntax of the language. By using the same approach in every Java class, we make sure that neither ourselves or other programmers ever have to spend any mental effort on thinking about what tools like this should be called or how they should work.

At first, adding accessors and mutators for every instance variable may feel like extra work, but actually it is an example of being lazy in a programmer way: we write these very simple methods in a standard way when we create the program, and this helps insure us against having to do complex, difficult updating of a class later on.

Suppose the Job class in the example above was part of a larger program that you had to maintain. After the program had been in use for a year or so, management tells you that the program now needs to do everything in terms of yearly salary (maybe because there was ambiguity about which type of salary, maybe for legal reasons, maybe just because of management whim).

So we add a new yearlySalary variable with its own accessor and mutator, but we know we can't remove the old accessor and mutator for the old salary variable – putting them into the public interface meant we promised they would always be available.

If every part of the program that had to access the salary went through that accessor and mutator, then if we just rewrite the bodies of the old accessor and mutator for the old salary variable to use the new yearlySalary variable, we shouldn't have to make changes to any other code to meet the management's requirements.

```
// class Job
public class Job {
    private double salary;
    private double yearlySalary;

   // old mutator, rewritten for new implementation
    public double getSalary() {
        return getYearlySalary() / 12;
    }

   // old accessor, rewritten for new implementation
    public void setSalary(double newval) {
        if (newval >= 0) {
```

```
            setYearlySalary(newval * 12);
        }
    }

    // new accessor for new implementation
    public double getYearlySalary() {
        return yearlySalary;
    }

    // new mutator for new implementation
    public void setYearlySalary(double newval) {
        if (newval >= 0) {
            yearlySalary = newval;
        }
    }
}
```

Notice that in the code above, the old salary accessor and mutator are passing values on to the new yearlySalary accessor and mutator. We're future-proofing again. If next year we have to swap yearlySalary for something else, we will only have to update the yearlySalary accessor and mutator.

Constructors

When we create an object based on a class type, we use the new operator, followed by the name of the class with parens on the end. We also know that most things followed by parens are methods. And indeed, we are calling a method in lines like

```
Dog spot = new Dog(); // calling constructor method
```

This method is called the constructor, and its job is to set up the instance variables for the object.

We know that instance variables, unlike local variables, have default values (0, false, or null) but for most classes, we would like to start the instance variables as different values. Technically we could do this where we declare the instance variables for simple literal values, but standard Java style is to do this in the constructor.

The name of the constructor (as we can see when using new) is always the same as the name of the class. It has no return type, and cannot be called by using its name other than by using new.

```
public class Dog {
   private String name;
   private int age;

   // constructor
   public Dog() {
         // okay
         // name = "unnamed doggy";
         // age = 1;

         // better
         setName("unnamed doggy");
         setAge(1);
   }

   // accessors, mutators (no 0 or negative age), toString…
}

// in main
Dog d = new Dog()// d starts with "unnamed doggy" and 1
Dog(); // error - cannot call without new
```

In the default constructor, it is okay to assign values to instance variables directly using =, but it is almost always better to use the mutator[3]. Then there is one fewer line of code to change if we have to change that instance variable later as discussed earlier. The only strong reason to use direct assignment instead is in the unusual case where an instance variable can start as a value that it cannot then be changed back to later, then we can assign it directly in the constructor, but the mutator can keep it from ever being changed back once it has been changed to a different value.

To see what instance variables start as, Java programmers are used to looking at the constructor. So even if our class does want to use the universal defaults (0, false, null) we might explicitly set an instance variable to that value in the constructor to make it clear that we didn't forget to set it, we actually want that value.

Classes can actually have extra constructors to make it quicker to assemble a new instance of the class and set instance variables at the same time. These are often called

[3] NetBeans will mark this with a warning. Remember that you can usually ignore warnings. For once, this is a meaningful message, but one outside the scope of what we've talked about so far.

parameterized constructors, because they use parameters to allow passing in values, while the constructor without parameters is called the default constructor[4] or no-parameter constructor.

```
public class Dog {
   private String name;
   private int age;

   // default constructor
   public Dog() {
         setName(“unnamed doggy”);
         setAge(1);
   }

   // parameterized constructor… missing something
   public Dog(String nameVal, int ageVal) {
         setName(nameVal);
         setAge(ageVal);
   }
   // accessors, mutators (no 0 or negative age), toString…
}

// in main
Dog d = new Dog()// d starts with “unnamed doggy” and 1
d.setName(“Spot);
d.setAge(4);
Dog d2 = new Dog(“Fluffy”, 3) // d starts with “Fluffy” and 3
```

Now, instead of creating a Dog with the default values and then using mutators to set the instance vars, if we know the values we want when we create the Dog, we can do all of that in one line. If a class had dozens of instance variables, providing a parameterized constructor could make your main code much shorter and easier to write; taking the time to write a good parameterized constructor is a good example of programmer laziness – put in the time to write a few lines of constructor once to save many lines of main code later.

[4] Other backgrounds use “default constructor” to refer to the invisible constructor that does nothing.

Notice that the parameterized constructor definitely needs to use the mutators, since we can't control what values are going to be passed in as arguments, so we need to make sure the mutators get to check the values.

But if the value passed to the parameterized constructor is invalid, in the code above, the instance variable would end up with the general default value (0, false, null). What if that value is also invalid for that variable in our class? We need to make sure that instance variables *start* with valid values, before we try to set them to the values given as parameters.

We know that the default constructor will set instance variables to valid values, so what we would like to do is call the default constructor, but we know we can't call it by name without new, and calling it with new would be creating *another* Dog. So there is special syntax to call one constructor from another – a new usage of `this`. One constructor can call another constructor in its first line by using `this()` this is called *chaining constructors*.

```
public class Dog {
   private String name;
   private int age;

   // default constructor
   public Dog() {
         setName("unnamed doggy");
         setAge(1);
   }

   // parameterized constructor, corrected
   public Dog(String nameVal, int ageVal) {
         this(); // call default constructor
         setName(nameVal);
         setAge(ageVal);
   }
   // accessors, mutators (no 0 or negative age), toString…
}

// in main
Dog d = new Dog()// d starts with "unnamed doggy" and 1
d.setName("Spot);
d.setAge(4);
Dog d2 = new Dog("Fluffy", 3) // d2 starts with "Fluffy" and 3
```

```
Dog d3 = new Dog(“Muffy”, -10) // d3 starts with “Muffy” and 1
```

Now, every dog starts with the values from the default constructor, and then the parameterized constructor tries to change it from there, but the mutator might stop an invalid value from getting through.

We can have multiple parameterized constructors in a class as long as the parameter lists are different in number and/or types of parameters. Choosing parameterized constructors is part of the design of a class, you should provide parameterized constructors to cover the common cases for what instance variable values are likely to be known when an object is created.

Suppose that half the time we know all the information for a Dog when it is created, but a significant portion of the time we just know the name. Then we might have

```
public class Dog {
   private String name;
   private int age;

   // default constructor
   public Dog() {
         setName(“unnamed doggy”);
         setAge(1);
   }

   // parameterized constructor, just name
   public Dog(String nameVal) {
         this(); // chain back to default
         setName(nameVal);
   }

   // parameterized constructor, name and age
   public Dog(String nameVal, int ageVal) {
         this(nameVal); // chain back to name-only
         setAge(ageVal);
   }
   // accessors, mutators (no 0 or negative age), toString…
}

// in main
```

```
Dog d = new Dog()// d starts with “unnamed doggy” and 1
d.setName(“Spot);
d.setAge(4);
Dog d2 = new Dog(“Fluffy”, 3) // d2 starts with “Fluffy” and 3
Dog d3 = new Dog(“Muffy”) // d3 starts with “Muffy” and 1
```

Notice that now the 2-parameter constructor calls the 1-parameter constructor which calls the default constructor. They are linked like a chain, which is why it is known as chaining constructors.

If you think about the code we have written previously, you might ask the obvious question: we haven't been *writing* constructors, and yet we've been *calling* constructors. Even if we haven't written a constructor in the Dog class, we could still say

Dog d = new Dog();

So what constructor is that code calling? If you create no constructors in your code, Java provides an invisible constructor that just does nothing. As soon as you do write at least one constructor yourself, Java does not do this. This means that if you write a parameterized constructor but no default, then your class just doesn't have a default constructor – you can't create an instance of your class at all without providing values up front.

```
public class Dog {
   private String name;
   private int age;


   // parameterized constructor, just name
   public Dog(String nameVal) {
         setName(nameVal);
   }

   // parameterized constructor, name and age
   public Dog(String nameVal, int ageVal) {
         this(nameVal); // chain back to name-only
         setAge(ageVal);
   }
   // accessors, mutators (no 0 or negative age), toString…
}
```

```
// in main
Dog d = new Dog()// error - no default dog can be created!
Dog d2 = new Dog("Fluffy", 3) // d2 starts with "Fluffy" and 3
Dog d3 = new Dog("Muffy") // d3 starts with "Muffy" and 1
```

This can be used in the rare case where in the process of designing a class we realize you should never be able to create an instance without certain up-front values. For instance, we might decide that a bank account cannot be created without the name and ID number of the owner of the account.