Java Foundations Notes

Chapter 5 (part 2) - Classes

CSC110

# Object-Oriented Programming

Java is an **Object-Oriented Programming** language. (OK, that isn’t 100% true, but it is close enough for our purposes.) The fundamental entity is the “**object**”

An object has some information (**state**) & some operations (**behaviors**) and usually represents some real-world entity such as:

* A particular student in a class
* A window in a GUI
* A character in a game

Objects should handle their own processing and data management.

Object-Oriented languages vary widely, but typically the contain the following features, often called “The Four Pillars of Object-Oriented Programming”:

* **Abstraction** - shifting the focus on what an object does instead of the details of its implementation
* **Encapsulation** - separating the interface from the implementation
* **Inheritance** - subclasses can include attributes & methods from a superclass
* **Polymorphism** - methods get called based on the type of the object no matter what the type of the reference is (late-binding)

The explanations above are by necessity fairly hand-wavy for now. We will discuss each of these in more detail later.

In Object-Oriented Programming there are usually multiple ways to represent a problem and there is often no “one right answer”. In fact, in many cases OOP itself is not always the answer (though for purposes of this class it will nearly always be). In practice, there are usually multiple ways to solve a problem, with a few of them being good choices. There are also lots of ways to poorly solve a problem, which is why experience is your best friend when it comes to programming. The more you practice the better you will get.

## Objects

Objects have a state, which is the values of the data members (also called **fields** or **instance variables**), which is what the object knows about itself. In general, an object should not allow external entities to change its state.

Objects also have behaviors, which are the methods (also called **members** or **functions**) it contains. The behavior of an object describes what the object can do. The behavior of an object may change its state. In fact, objects should *only* allow their state to be changed through their behaviors.

Think of calling a method as “sending a message” that asks the object to do something. The message contains the operation’s name and arguments.

The client doesn’t care how the message is handled, only that it produces an expected result. Thinking this way will help you better design the classes in your program.

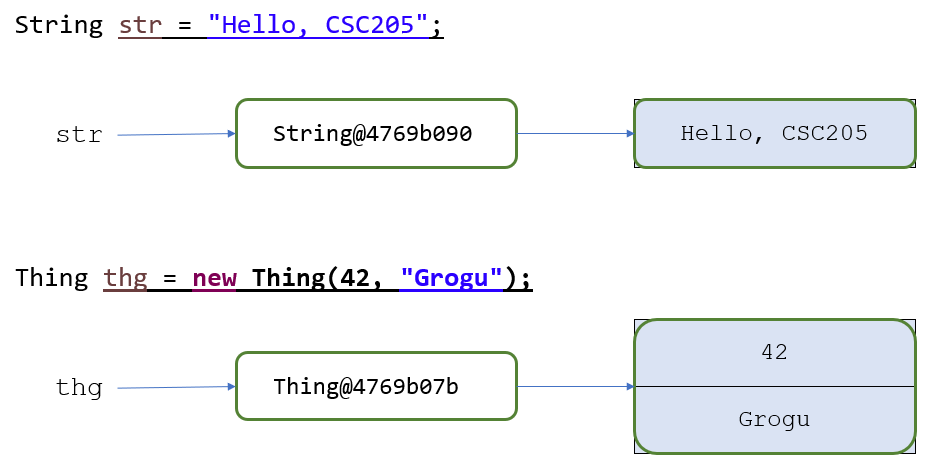
Creating an object is called **instantiation**. In Java, the new operator creates a new object. Each object is an **instance** of a particular class.

#### Objects vs Classes

A **class** represents a abstract concept, while an object is the realization of a class

We **instantiate** an object of a specific class. There can be multiple objects of a given class, but each object is an instantiation of a single class

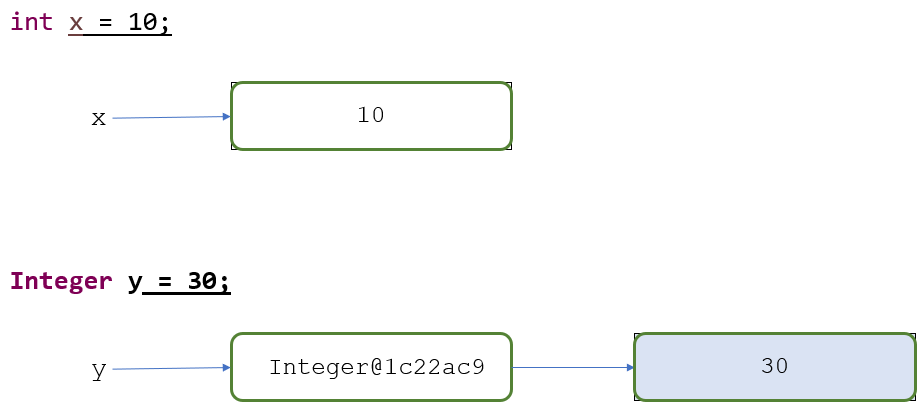
An **object reference variable** contains a reference to an object rather than the variable itself. A class name used as the type when declaring an object reference variable. The example below shows two object reference variables: str is a String object reference variable that refers to a string object with the contents “Hello, CSC205” and thg is a Thing object reference variable that refers to a Thing object with the contents {42, “Grogu”}. Note that the actual str & thg *variables* only hold a reference to the objects, *not the contents of the objects themselves!*



Java has a corresponding **wrapper class** for each primitive type. For example:

* int → Integer
* char → Character
* double → Double

Below is an example of an int primitive variable and an Integer object reference variable.



These wrapper classes have static methods that help manage objects of that type. Java performs automatic **auto-boxing** to convert primitive types to the corresponding wrapper object type.

## Constructors

In Java, each class can have a **constructor** that initializes objects of that class when they are instantiated. The constructor is a method that has the same name as the class and no return value. Constructors can be overridden, and you can call the super class’s constructor using **super()**.

## Accessors & Mutators (Getters & Setters)

While a class should not allow public access to its data, in many applications the client may want to view or change the data. **Accessor** (getter) & **mutator** (setter) methods can provide this capability. This allows the object to still supervise how its state is managed without having to provide direct access to the data. Keep in mind, your class does not have to provide accessors and mutators for all of its attributes.

One common mistake is for the class writer to expect the client to use mutators to initialize an object. While this works, it forces the client to know what fields need to be set up - a better way is to have all initialization done in the constructor.

Bad:

##### Student newGuy = new Student();

##### newGuy.setName("Blanche Mayne");

##### newGuy.setAge(20);

##### newGuy.setID("815162342");

Better:

##### Student newGuy = new Student("Lucas Sankey", 19);

Notice no ID is passed to the constructor. That is because something like “generating an ID” is best left to the class and not handled by the client at all.

## The **toString** Method

The Java toString() method is used to return a String representation of an object. It is defined in the Object class and should be overridden in user-defined classes. This means that you should implement a toString method in the classes you define.

By implementing the toString method, you are providing an easy way to return the state of an object as a String.

Here’s a simple example:

#### public class Student {

#### 

#### //instance variables - private to ensure encapsulation

#### private String name;

#### private int id;

#### 

#### public Student(){ //default constructor

#### name = "unknown";

#### id = 0;

#### }

#### public Student(String someName, int someId) ( //overloaded constructor

#### name = someName;

#### id = someId;

#### }

#### 

#### /\*returns a String with the current values for all instance variables.

#### This is also called the state of an object

#### How you format your String is up to you, this is just one example

#### Since you are overriding a method in the Object class, the method

#### signature must be identical to what is given here. \*/

#### public String toString() {

#### return ("[name: " + this.name +

#### " | id: " + this.id + "]");

#### }

#### }

Now, take a look at the tester and how toString is used in a System.out.println() statement.

#### //An example to show how to use toString method

#### public class TestStudent {

#### 

#### public static void main(String[] args) {

#### 

#### //Create a Student object using the default constructor

#### Student s1 = new Student();

#### 

#### //display the state of s1 using toString

#### System.out.println("The current state of Student s1: " + s1.toString());

#### 

#### //Create another Student object using the overloaded constructor

#### Student s2 = new Student("Luke Skywalker", 34892);

#### System.out.println("The current state of Student s2: " + s2.toString());

#### 

#### //notice that toString is implicitly called in this next example

#### System.out.println("Another example: The current state of Student s2: " + s2);

#### }

#### }

Evaluate the output generated. The blue text has been generated by toString().

**The current state of Student s1: [name: unknown | id: 0]**

**The current state of Student s2: [name: Luke Skywalker | id: 34892]**

**Another example: The current state of Student s2: [name: Luke Skywalker | id: 34892]**

## Special References

Every Java object contains some special references. One of these references - “**this**” - refers to the object itself. So, if an object has a string member called address and a double member called value, you can refer to those two values as this.address and this.value.

In most cases there is no reason to use the “this” reference (though it isn’t wrong to do so), but it allows an object to refer to itself, such as calling another overloaded constructor or referring to its members when they have been **shadowed** by a local variable name or parameter. This reference can be useful to make methods more clear, especially when another object of the same type is involved, such as when a class has a method that takes an object of that same class as a parameter (as with compareTo).

For example, suppose we have an Account class with the following members:

##### private String name;

##### private int credits;

##### private int accountNumber;

And suppose one of the constructors is as follows:

##### public Account(String name, int credits) {

##### super();

##### this.name = name;

##### this.level = credits;

##### this.accountNumber = generateAccountNumber();

##### }

Here, the parameter name shadows the member name. So when we refer to simply “name” in the method we are referring to the String parameter name. In order to refer to the member name we have to use the “this” reference. There was no technical reason to use the “this” reference for level and account number, but it was done for consistency.

As another example, suppose our class above had a method like below:

##### public boolean verify(Account otherAccount) {

##### if (this.level != otherAccount.level) {

##### return false;

##### }

##### else {

##### return (this.accountNumber > otherAccount.accountNumber);

##### }

##### }

In this case, we are passing an Account object in as a parameter to our verify method, so it would be called something like this:

##### acct1.verify(acct2);

We have two objects - acct is the account the method is being called on and acct2 is the parameter. So we have two sets of members. In the method above we use this to refer to acc1 members and otherAccount to refer to acc2 members.

Note that this cannot be used in a static method since it refers to a specific instance. There is also a super reference that is used to refer to the parent class. We will discuss that reference in the next module when we talk about inheritance.

# Considerations When Designing Classes

## Static Class Members

Static class variables have one value shared among all objects of that class. So if it is updated in one object, all objects of that class will see it. Static class methods can be called without instantiating an object, but they can only access static data members. Deciding which members to make static is an important design decision

Static members are associated with *the class as a whole*.

Non-static members are associated with *individual objects*.

Suppose we have a class called Static with the following data members:

**private static int *staticVariable***

**private int nonStaticVariable;**

**private String name;**

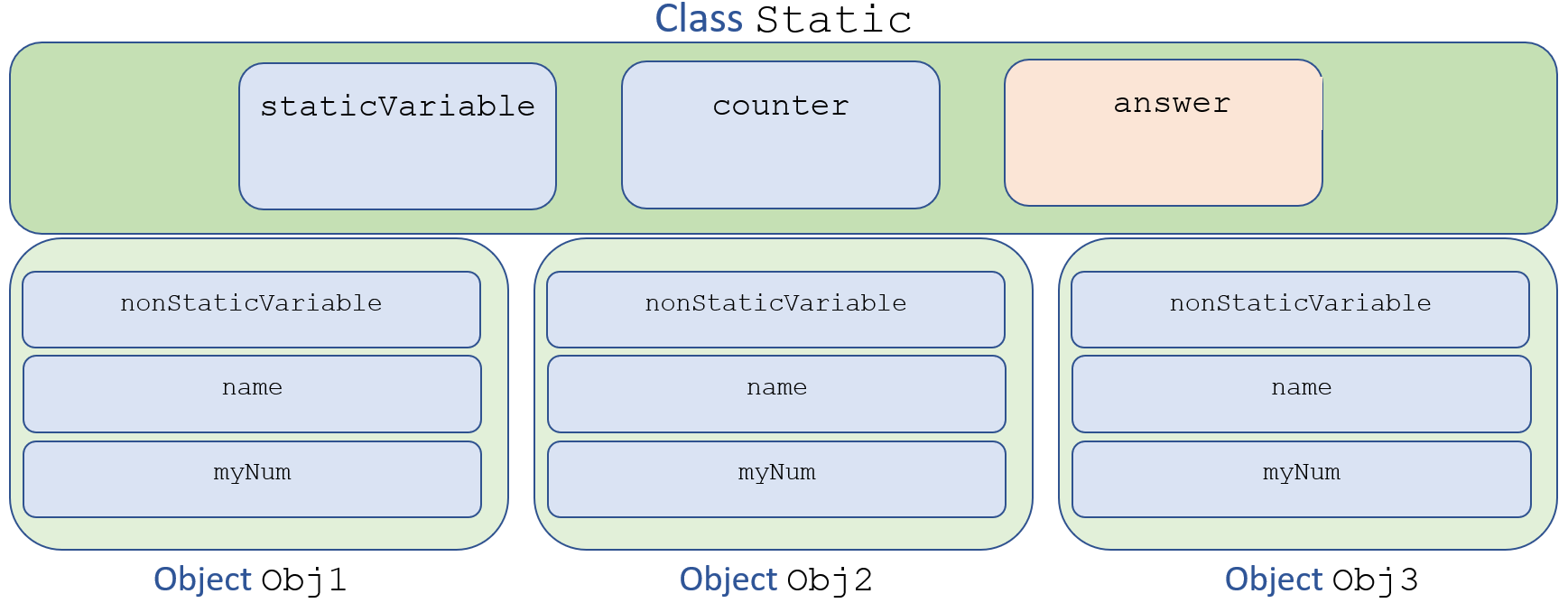
**private static final int *answer = 42;***

**private int myNum;**

**private static int *counter = 0;***

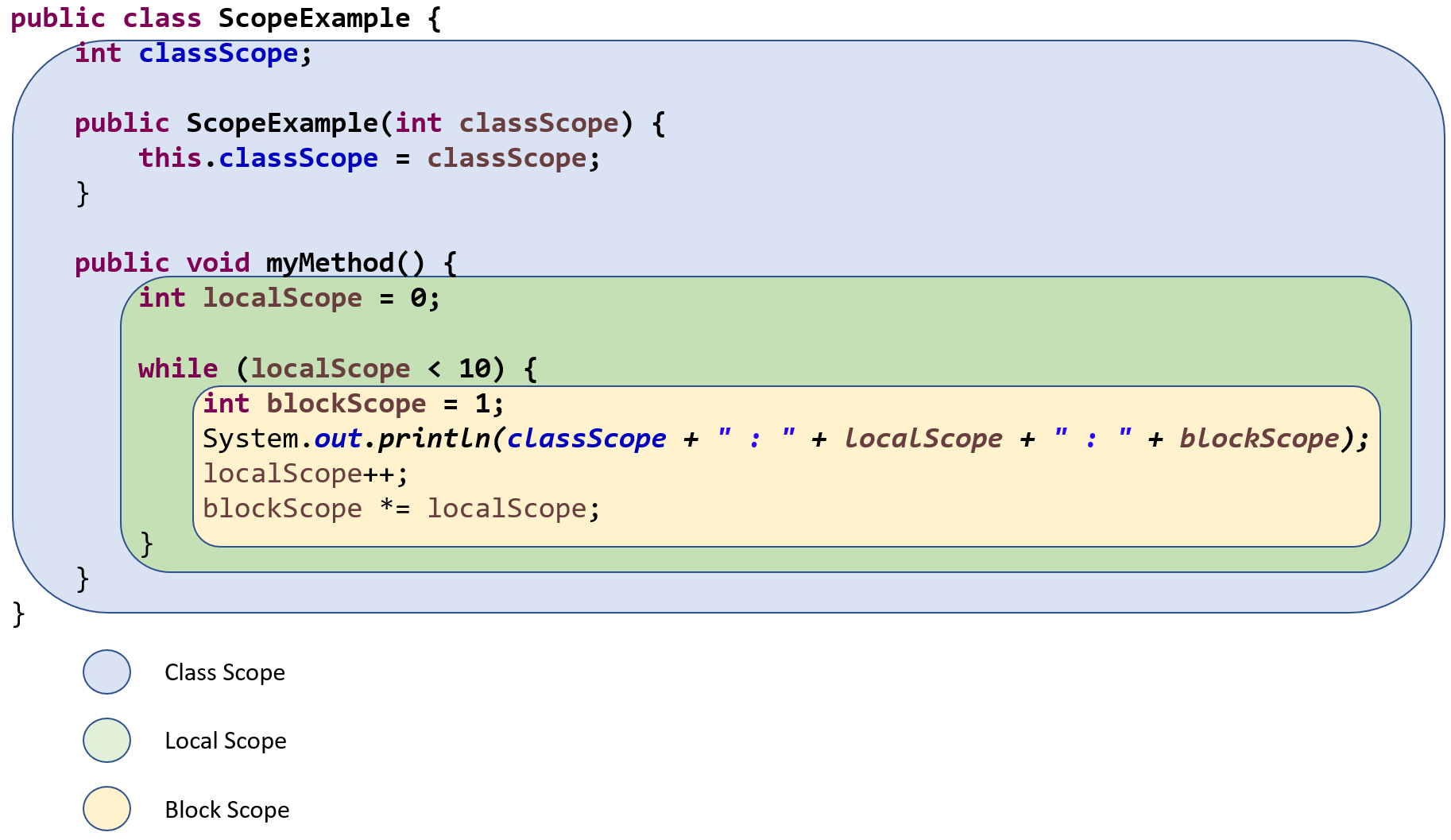
The objects of the class share the static members (***staticVariable***, ***answer***, ***counter***), so there is one copy of each of those members - changes to those static members in one object affect every other object. The non-static members (**nonStaticVariable**, **name**, **myNum**) are not shared, so each object gets its own copy.

Suppose we have three objects of type Static. Here is what that would look like:



## Data Scope

The **scope** of data is the part of the program in which it can be referenced (where it is visible). Data members of a class can be referenced by all methods in that class (so you should limit their use to only those variables that have use for all the methods in the class). Local variables in a method can only be referenced in that method. Data declared in a code block can only be referenced in that block. The example below shows how scope works in Java. Note that since blockScope has block scope it will always be 1 when we print the line of output!



## Visibility Modifiers

Java provides several **visibility modifiers:**

* **public** members are accessible to anyone
* **private** members are not accessible outside the class, even to subclasses
* **protected** members are public to subclasses, but private to everyone else
* **final** members cannot be overridden by subclasses

In general, data members should be private. The class should not allow any outside entity to change its state (the values of its data members) directly. The only way to change the state of an object should be through public methods. Keep in mind that methods can also be private, though only methods that are not intended to be used by the client should be made private. Protected members are used when we want to allow child classes access to them, but not make them publicly available.

## Class Relationships

Classes can be related in multiple ways. Suppose we have two classes Class1 and Class2.

* **Dependency** is when Class1 uses Class2. For example, a BankAccount class may use the java.util.Math class to perform calculations.
* **Composition** (Aggregation) is when Class1 has an instance of Class2. We call this a “has-a” relationship.
* **Inheritance** is when Class1 is a Class2, for example a Bird is an Animal, a Magician is a Character, a Truck is a Vehicle.

Be careful not to confuse inheritance with instantiation. Inheritance deals with the relationships between classes, while instantiation is creating a specific object of a class. So inheritance is used to indicate that a CliffTroll is a Troll, but we instantiate a CliffTroll object with the name Nero if Nero is an actual CliffTroll.

We will discuss inheritance & composition more later in the semester.

# More About OOP

## Abstraction

*“In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from the recognition of similarities between certain objects, situations, or processes in the real world and the decision to concentrate on these similarities and to ignore, for the time being, their differences.”* - C.A.R. Hoare

When we define a class we need to be concerned with the details of variables and methods. However, when we *use* an object we should only be concerned with its *behaviors* - the specifics of the internal variables and methods should be of little concern to us.

In discussions about Object-Oriented Programming, abstraction and encapsulation are often confused due to them being very closely related. In some ways, in OOP you can think of abstraction as a general concept of describing some object in simpler terms (as opposed to the details of its implementation), whereas encapsulation is the technique that is used to achieve abstraction.

Don’t get too hung up in the differences. The important idea with abstraction is that it lets you think of your program in simpler terms than the actual thing you are trying to represent. For example, if you want to go to Dallas, you call the goToDallas() method - whether you fly, drive, or walk isn’t necessarily a concern, your goal is to get there.

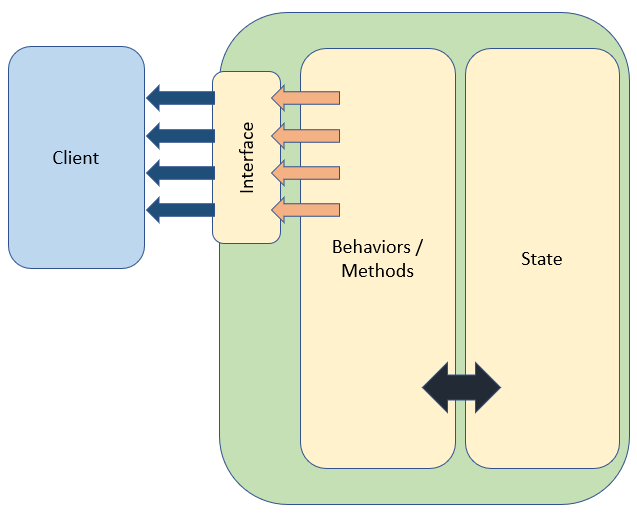
## Encapsulation

*“The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.”*

* + Grady Booch in *Object-Oriented Analysis and Design with Applications*

The idea behind encapsulation is to bring your abstractions together in a single unit. This allows an object to protect its data (though not all OOP languages provide this, as in Python). The users of a class only need to know the services that the object provides - its **interface** or **API**. (Note that in this context we are referring to the general idea of an interface, not the Java interface feature.) The specifics of how the parts of the object interact are encapsulated inside the object. The objects interface provides the behaviors the **client** (user of the object) needs.

From the client’s perspective, the object can be thought of as a black box. The client invokes the interface methods of the object to generate behaviors from the object, but the object still manages its data. Note that the behavior does not necessarily have to provide external access to all its behaviors.



### Encapsulating Behavior

A better way to think of encapsulation is that you should strive to encapsulate behavior, not just data. For example:

setAccount(1000000) vs deposit(1000000)

setDecor() vs decorate()

The user of the class shouldn’t be making decisions about how the state gets changed! Instead of setters that simply set the value of an attribute based on the client's demands, think of behaviors that can be provided as methods.

# Designing Classes

When designing classes, ask yourself two questions:

* What states does my class need to represent?
* What are the behaviors my class needs to implement?

You should always strive to make your classes reusable. It is possible all clients will not use each behavior or state of the class. It is usually better to write your class so that you can add additional methods in the future rather than trying to anticipate everything your class will ever need.