MC322 - Object Oriented Programming
Lesson 2.1
The Anatomy of a Class

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Object-Oriented Design

- **No Isolation:** No matter how well you think out the problem of what should be an interface and what should be part of the implementation, the bottom line always comes down to how useful the class is and how it interacts with other classes.
- **Contextual Design:** A class should never be designed in a vacuum, for as might be said, no class is an island. When objects are instantiated, they almost always interact with other objects.
- **Hierarchy and Composition:** An object can also be part of another object or be part of an inheritance hierarchy.

Note

This class is meant for illustration purposes only. Some of the methods are not fleshed out (meaning that there is no implementation) and simply present the interface—in part to emphasize that the interface is the primary part of the initial design.

The Name of the Class

Importance of Class Names

- **Identification:** The name of the class is important to identify the class itself.
- **Descriptiveness:** Beyond simple identification, the name must be descriptive. It provides information about what the class does and how it interacts within larger systems.
- **Language Constraints:** The choice of a name is important considering language constraints. For example, in Java, the public class name must be the same as the file name. If the names do not match, the application won't work.

```java
public class Cabbie {
}
```

Using Java Syntax

Remember that the convention for this book is to use Java syntax. The syntax will be similar but somewhat different in C#, .NET, VB .NET, or C++, and totally different in other object-oriented languages such as Smalltalk.

Comments

Comments in Different Languages

Regardless of the syntax of the comments used, they are vital to understanding the function of a class. There are two kinds of comments in Java, C .NET, and C++.

The Extra Java and C# Comment Style

In Java and C, there are actually three types of comments. In Java, the third comment type (** **) relates to a form of documentation that Java provides. We will not cover this type of comment in this book. C provides similar syntax to create XML documents.

The first comment is the old C-style comment, which uses /* (slash-asterisk) to open the comment and */ (asterisk-slash) to close the comment.

```
/*
This class defines a cabbie and assigns a cab
*/
```

The second type of comment is the // (slash-slash), which renders everything after it, to the end of the line, a comment.

```
// Name of the cabbie
```

Attributes

Attributes in Object-Oriented Programming

Attributes represent the state of the object because they store the information about the object. For our example, the `Cabbie` class has attributes that store the name of the company, the name of the cabbie, and the cab assigned to the cabbie. For example, the first attribute stores the name of the company:

```
private static String companyName = "Blue Cab Company";
```

Note here the two keywords `private` and `static`. The keyword `private` signifies that a method or variable can be accessed only within the declaring object.

Hiding as Much Data as Possible

All the attributes in this example are private. This is in keeping with the design principle of keeping the interface design as minimal as possible. The only way to access these attributes is through the method interfaces provided (which we explore later in this chapter).

User Attributes in Cabbie Class

Static Keyword: The static keyword signifies that there will be only one copy of this attribute for all the objects instantiated by this class. This is a class attribute.

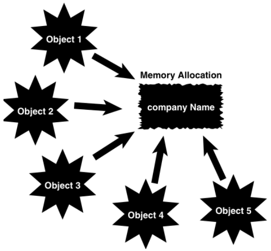
```java
private static String companyName = "Blue Cab Company";
```

Name Attribute: The second attribute, `name`, is a string that stores the cabbie's name. This attribute is also private so other objects cannot access it directly. They must use the interface methods.

```java
private String name;
```

myCab Attribute: The `myCab` attribute is a reference to another object. The class, called `Cab`, holds information about the cab, such as its serial number and maintenance records.

```java
private Cab myCab;
```



Object memory allocation.

Passing a Reference

Another object likely created the Cab object. Thus, the object reference would be passed to the Cabbie object. However, for the sake of this example, the Cab is created within the Cabbie object. Likewise, for this example, we are not interested in the internals of the Cab object.

Constructors

Constructors in Cabbie Class

This Cabbie class contains two constructors. We know they are constructors because they have the same name as the class: Cabbie.

Default Constructor:

```
public Cabbie() {
    name = null;
    myCab = null;
}
```

Technically, this is not a default constructor (but has a similar behavior). The default constructor is only provided if you provide no constructors in your code. In this constructor, the attributes Name and myCab are set to null: name = null; myCab = null;

The Nothingness of null

In many programming languages, the value null represents a value of nothing. This might seem like an esoteric concept, but setting an attribute to nothing is a valuable programming technique. You can check whether an attribute has been properly set by setting the attribute to null (a valid condition).

Initializing Attributes in Constructors

As we know, initializing attributes in the constructors is always a good idea. In the same vein, it's a good programming practice to test the value of an attribute then to see whether it is `null`. This can save you a lot of headaches later if the attribute or object is not set correctly. An exception might be generated if you treat an uninitialized reference as if it were properly initialized.

The second constructor provides initializes the `Name` and `myCab` attributes:

```
public Cabbie(String iName, String serialNumber) {
    name = iName;
    myCab = new Cab(serialNumber);
}
```

In this case, the user would provide two strings in the constructor's parameter list to properly initialize attributes. Notice that the `myCab` object is instantiated in this constructor:

```
myCab = new Cab(serialNumber);
```

Accessors

Encapsulation and Accessor Methods

The attributes are usually defined as private so that other objects cannot access them directly. But isn't it necessary to inspect and sometimes change another class's attribute? The answer is yes, of course. Sometimes, an object needs to access another object's attributes; however, it does not need to do it directly.

A class should be very protective of its attributes for several reasons; the most important ones boil down to data integrity and efficient debugging.

Assume that there is a bug in the Cab class. You have tracked the problem to the Name attribute. Somehow, it is getting overwritten, and garbage is turning up in some name queries. Name were public you would have to search through all the possible codes, trying to find places that reference and change Name. However, if you only let a Cabbie object change its name, you'd only have to look in the Cabbie class.

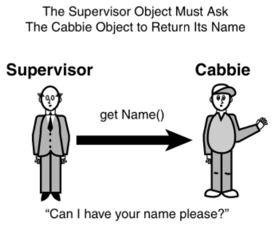
An accessor is a type of method that provides this access. Sometimes accessors are referred to as getters and setters, and sometimes they're simply called `get()` and `set()`. By convention, in this book we name the methods with the `set` and `get` prefixes, as in the following.

Accessor Methods and Data Integrity

```java
// Set the Name of the Cabbie
public void setName(String iName) {
    name = iName;
}
// Get the Name of the Cabbie
public String getName() {
    return name;
}
```

In this code snippet, a Supervisor object must ask the Cabbie object to return its name. It is good because you might have a `setAge()` method that checks to see whether the age entered was 0 or below. In general, the setters are used to ensure data integrity.

This is also a security issue. You may have sensitive data, like passwords or payroll information, which you want to control access to.


The Supervisor Object Must Ask
The Cabbie Object to Return Its Name
Supervisor
Cabbie
get Name()
"Can I have your name please?"

Asking for information.

Objects and Static Attributes

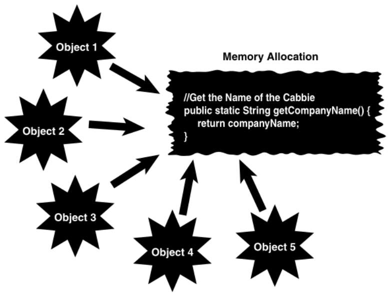
getCompanyName method is declared static as a class method and attribute companyName is also declared static. Like an attribute, a method can be declared static to indicate only one copy of the method for the entire class.

Objects

Actually, there isn't a physical copy of each non-static method for each object. Each object would point to the same physical code. However, from a conceptual level, you can think of objects as being wholly independent and having their own attributes and methods.

Static Attributes

If an attribute is static, and the class provides a setter for that attribute, any object that invokes the setter will change the single copy. Thus, the value for the attribute will change for all objects.



Method memory allocation.

```
// Get the Name of the
    Cabbie
public static String
    getCompanyName() {
    return companyName;
}
```

Interfaces and Implementation

Public Interface Methods

Both the constructors and the accessor methods are declared public and are part of the public interface. They are singled out because of their specific importance to the class's construction. However, much of the real work is provided in other methods. The public interface methods tend to be very abstract, and the implementation tends to be more concrete.

For this class, we provide a method called `giveDestination()` that is the public interface for the user to describe where she wants to go:

```
public void giveDestination (){
}
```

What is inside of this method is not important at this time. The main point here is that this is a public method and part of the class's public interface.

Private Implementation Methods

Some methods in a class may be hidden from other classes. These methods are declared as private:

```
private void turnRight(){
}
private void turnLeft() {
}
```

These private methods are meant to be part of the implementation, not the public interface. These methods are only called internally from the class itself.

```
public void giveDestination (){
    // some code
    turnRight();
    turnLeft();
    // some more code
}
```

For example, you may have an internal method that provides encryption that you will only use from within the class. In short, this encryption method can't be called from outside the instantiated object itself.

Conclusion

In this chapter, we have delved into a class and described the fundamental concepts necessary for understanding how a class is built. Although this chapter takes a practical approach to discussing classes, Chapter 5, ”Class Design Guidelines,” covers the class from a general design perspective.

This lesson is based on Chapter 4 of ”The Object-Oriented Thought Process”.

MC322 - Object Oriented Programming
Lesson 2.1
The Anatomy of a Class

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP