Lecture Notes, CS345, Winter 2018

Introduction to Object-Oriented Programming

Part II: Definitions and Terminology

What exactly is an Object?

An object has

1. data or attributes
2. behavior: what can an object do? You can think of these as:
   - a collection of operations on the object, or
   - a collection of methods on the object, or
   - a collection of messages understood by the object.

Here's picture of the message idea:

Characteristics of a method or operation:

- A name
- Parameters of the method:
  - the number of parameters,
  - the types of the parameters,
- the type(s) of the values, if any, that are returned

What is a Class?

- "A blueprint for an object."
- Alternatively, a template for an object.
- Alternatively, a cookie cutter for an object.

In Java when you code:

```
class C { ... }
```

you are writing the description of (potentially) many objects. When you code:

```
new C(...)
```

you are asking for the *instantiation* or *construction* of a single new object according to the blueprint or template that you defined when you coded the class.

Here are some things described by the class:

- The attributes: name, type, whether they are *public*—accessible by any client—or *private*—only accessible from inside the object, or something in between.
- The methods: name, parameters, return values, whether they are *public*, *private*, or … .
- *Constructors*: a specialized kind of method used to initialize new objects. Note that, in spite of the name, the purpose of a constructor is to initialize a new object that has already been created by the programming language.

What is a Type?

A type is a collection of behavior that can be accessed and used by client code. In this sense *type* has the same meaning as the word type in *Abstract Data Type*.

- There is a many-to-many relationship between types and classes. A given type can be implemented by many classes and a given class can provide more than one type.
- A given object has exactly one class. However, a given object can have many types.

In Java, when we declare an object:

```
T obj = … ;
```

`T` is a type. `T` may also be a class but is not required to be. The declaration specifies what may be done with `obj` within the scope of the declaration—that part of the code where the declaration of `obj` is visible.

Information Hiding and Encapsulation

An object does not have to reveal all its attributes and behaviors. This is good!

Information Hiding is a software engineering principle that calls for hiding design decisions so that they can be changed without impacting the rest of the program.

Encapsulation is a programming language mechanism that provides for bundling data and function together so that the only aspects that are exposed are those that are intended to be used by client code. From the perspective of client code, the exposed aspects can be thought of as what the object "does" or what the object is "responsible" for.

Some authors consider information hiding and encapsulation as being synonymous. Others consider information hiding as the principle and encapsulation as the programming language mechanisms, like *public* and *private* keywords in Java, that support information hiding.

Good design: An object should only reveal only those aspects that client code must access to interact with the object. That is, anything used only by the object internally should be *private* to the object. The opposite of private is *public*. Public attributes and methods are available to any code using of the object. Note: public and private are two of Java's four (Yes, four!) different levels of access.

Heuristic: It is much easier to change something from private to public—increase accessibility—than to do the reverse. For this reason, if you are unsure how much access should be provided to some aspect of an object, you should use the most restrictive accessibility you can until you have an example that shows otherwise. That is, if you don't know whether it should be public or private, use private until you know that's wrong.

Interface

The design of a class specifies the *interface* for the operations that can be performed on the class and for *constructing* an instance of the class. Any behavior that is invoked on the class is invoked via one of the provided interfaces.

Getters and Setters

A special group of operations is *getters* and *setters*. A *getter* is a method that returns the value of an attribute. A *setter* is a method that sets the value of an attribute. In Java, a getter and setter for an attribute named `x` of type `T` would look like:

```
private T x; // clients cannot access or modify x directly

public T getX() { return x; }

public void setX(T value) { x = value; }
```

In the above code, the private attribute `x` is entirely optional! We can eliminate `x` and replace the code in `getX` and `setX` with code that makes it look as if there is an attribute `x`. Client code would not need to change to accommodate this. Note that this approach allows the object to maintain control of access to its attributes. For example, you could make `x` a read-only attribute by not providing the setter.

In Java, all attributes of an object should be declared as `private`. Getters and setters should always be used in Java. (This may or may not be true for other languages.)

In Java, a `public` attribute breaks information hiding. Making the attribute public exposes the design decision that there is an actual attribute `x`, as opposed to operations that provide the behavior of an attribute `x`.

In Java, converting attribute access to getter and setter access cannot be done without affecting the client code. In Java, the expression `obj.x` can only refer to an attribute named `x` of the object `obj`.

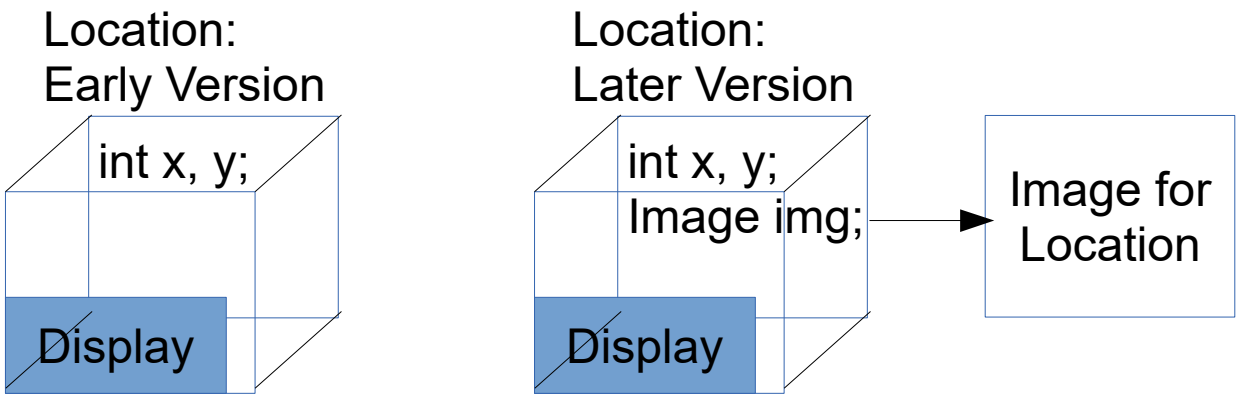
Other languages, such as C# and Python, allow you to make this conversion without changing the client code. The compiler and run time in these languages allow you to specify that the expression `obj.x` actually refers to getter and setter methods. In this case, what is written as attribute access is converted to calls to the appropriate getters and setters.

Implementation

How a class is implemented.

In addition to the behavior of a class—the operations that are available to clients of the class—the implementation of a classes can include:

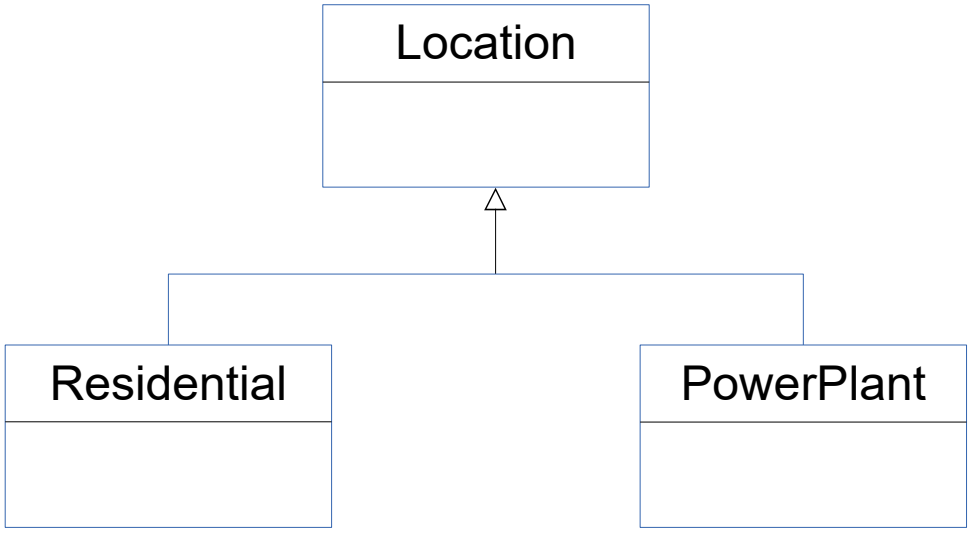
- private attributes (typically)
- additional methods that are not part of the external behavior (frequently)



Example: Consider a Location object. How do you display a Location object? Early in the development, you might draw a box at the object location, maybe with a label. Later, there might be an image that's displayed at the object location.

The important idea here is that there are two implementations of the Display function. Both implementations have the same interface. So, even though the implementation has been significantly changed, the client code does not have to change. Another way to say this is that there has been a change to the Location *class* but not the Location *type*.

Inheritance



Notation: This is an example of a *UML class diagram*. The boxes are classes. The name above the top line is the name of the class. The open delta on the end of the line indicates

an inheritance relationship where Location is the common *parent* and Residential and PowerPlant are *children*.

Idea of inheritance: The child inherits both interface and implementation from the parent. The child is then free to extend the interface and modify the implementation, within constraints.

Terminology:

- *parent* or *superclass*. Location is the parent or superclass of Residential.
- *child* or *subclass* or *derived class*. PowerPlant is the child or subclass of Location.
- *generalization*. The superclass is a generalization of its subclasses, so Location is a generalization of Residential and PowerPlant.
- *specialization*. The subclass specializes its superclass, so Residential is a specialization of Location.

IS-A Relationships

Sometimes we talk about subclass/superclass relationships as "IS-A" relationships. For example, Residential IS-A Location. However, IS-A does not necessarily imply inheritance. IS-A only implies inheritance of the interface. In order to say that Residential IS-A Location, we only require that the interface or external behavior or type of a Residential has to extend that of a Location. The implementation is internal to the class.

Another way to think about this: When we say that Residential IS-A Location, we are saying that anything that can be done with a Location object can be done with a Residential object. Any function operation that takes a Location object as a parameter will also work correctly with a Residential object. This means that not only must Residential objects have methods with names, parameters, and return types that match those of a Location, but, in addition, these methods must behave "the same" as those for a Location. "The same" is interpreted people: the developers of the software.

Polymorphism

In object-oriented languages, there are two concepts of polymorphism:

1. Multiple functions with the same name but different signatures. The actual function invoked is chosen based on **the declared *types* of the parameters as determined by the *compiler at compile time***.

   Example: In Java, `System.out.print` can print anything, both primitive and reference types. According to the documentation for `java.io.PrintStream`, there are nine versions of `print`, six for primitive types (`byte` and `short` use `int`), one for `Object`, one for `String`, and one for `char[]`.

2. (Object-Oriented definition) Multiple functions on different classes, all functions with the same name and signature. **The actual function invoked is *chosen at run-time* based on the actual *classes* of the parameter(s) as determined at run-time.**

Example: In Java, all classes inherit from the class `Object`. The `Object` class defines a method `public String toString()` that returns a printable description of the object. You can change how an object of any class is printed by defining a `toString` method for that class. When the program is run, the `print` function will find the appropriate `toString` method for the actual object that is passed to `print`. That `toString` method will be called to get the `String` that is output by that call to `print`.

Overloading and Overriding

For type 1 polymorphism, multiple functions which are distinguished by the compiler based on the type of the parameters, we say that the function is *overloaded*. So, the function `System.out.print` is overloaded

For type 2 polymorphism, when the implementation of a function in a child is used instead of the implementation of the "same" function in a parent, the function in the child is said to *override* the implementation in the parent. An implementation of `toString` overrides the default implementation in the class `Object`.

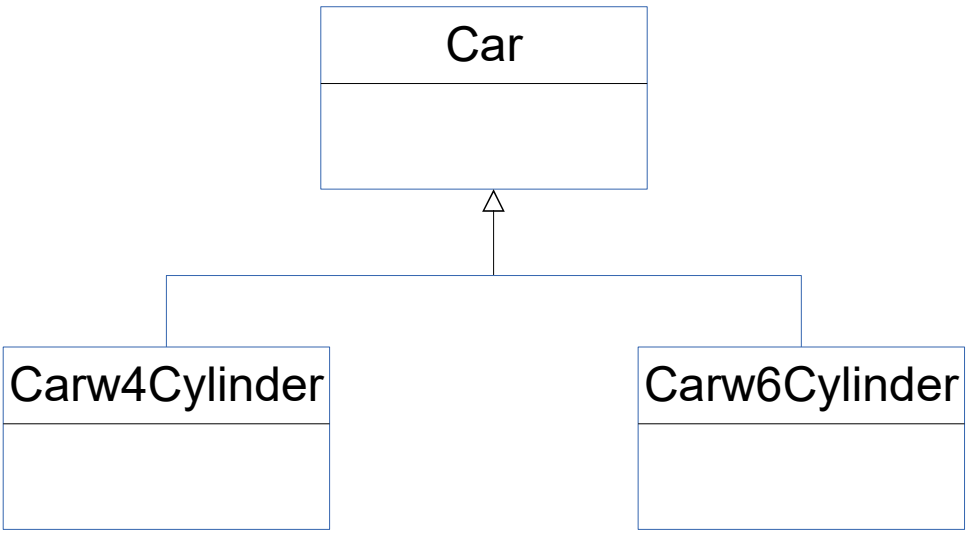
In Java, you can specify that one method overrides another by annotating the overriding method. For example:

```
@Override public String toString() { ... }
```

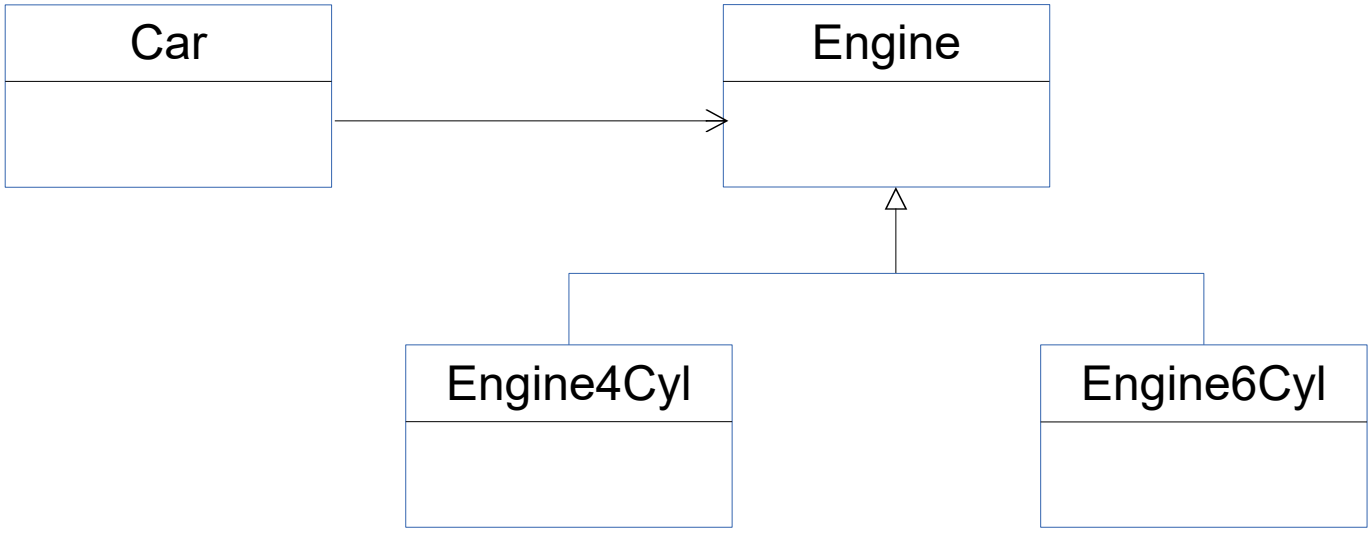
The symbol `@Override` is not required. However, if you include the annotation and the method is not overriding, the compiler will produce an error message. This annotation also acts as a way to document that the method is overriding.

Composition (HAS-A)

Composition is when one object "contains" or HAS-A another. For example consider cars with different kinds of engines. Here is an example of doing this using inheritance:



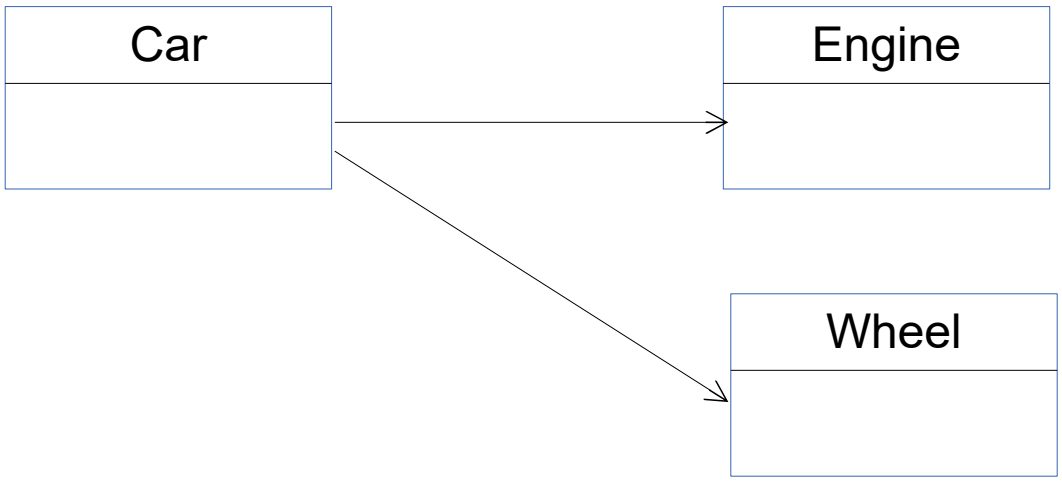
Here is the same idea using composition:



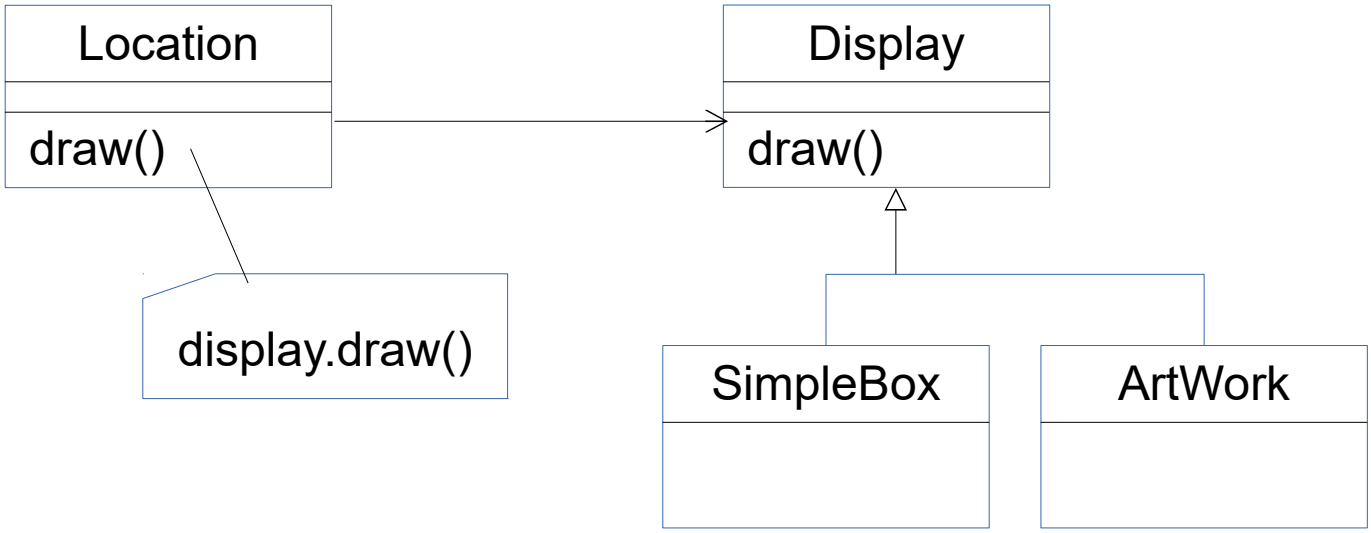
Notation: The line with the open arrow at the end says that a Car object contains a reference to an Engine object.

In this example, inheritance is used to model the different types of engines. However, the Car object includes the Engine object by reference. Using this model, we could, for example, change the kind of engine that is in our car or move Engines between Cars without having to create new Car objects.

In the next example, the Car object now has both Engine and Wheel objects. This is easy to model using composition. Attempting to model this using inheritance will result in multiple levels of inheritance, each level distinguishing some different attribute of the Car. This is known as a "nested generalization hierarchy" and should generally be avoided.



Finally, returning to our Location example. You could handle drawing of Locations using composition as follows:



In this diagram, the Location object refers to a Display object via the display attribute of a Location object. The draw method on Location simply calls the draw method on the associated Display object. The diagram shows two possible types of Display objects, one that draws a simple box and one that does a more complex drawing based on some complex art work.

This technique, of having a method on the owning object simply call a corresponding method on the owned object is quite common when using composition.

Final Thoughts

Analysis and Design Problem: Behavior

When creating an Object-Oriented program, a key problem we need to address is identifying the behavior we expect from each object. Once we understand these behaviors, we can go about writing the code that implements that behavior.

Ultimately, we will focus on separating the externally visible behavior from the internal problem of how we produce or implement that behavior.

Classic Object-Oriented Applications

The whole concept of object-oriented programming originated from two kinds of programs:

- Simulations
- Graphical User Interfaces (GUIs)

In both of these kinds of applications, there are many common behaviors that have to be implemented, for example, drawing a graphical object on the screen in a GUI. These kinds of applications benefit greatly from the object-oriented approach.

Conclusion

Key ideas:

- Object
- Class
- Type
- Information Hiding and Encapsulation
- Interface
  - Getters and Setters
- Implementation
- Inheritance
  - Super- and sub- classes
    - Superclass also known as parent
    - Subclass also known as derived class or child
  - Generalization and specialization
  - IS-A relationship
- Polymorphism
  - Overloading
  - Overriding
- Composition
  - HAS-A relationship