SE3430/5430 OOAD F13
Lecture Notes 03

Kun Tian

Object Orientation

- What is an object? (Object Orientation, or OO)
- How does OO relate to "building a good system"?
- What is inheritance?
- What is polymorphism and dynamic binding?

Object

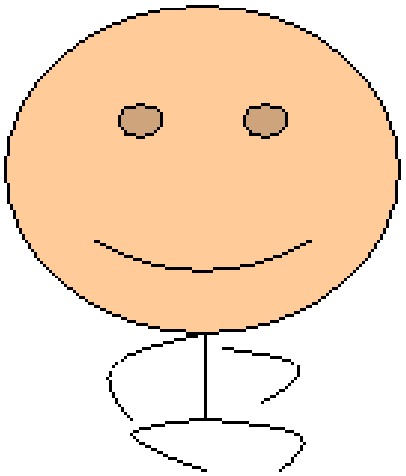
- In computer science, an object is a location in memory having a value and referenced by an identifier. An object can be a variable, function, or data structure.
- Objects in OO basically are data structures combined with the associated processing routines.
- Conceptually, an object is a thing you can interact with: you can send it various messages and it will react.
- An object is a "thing". "It is a thing in the system." it is the system representation of a thing, be it physical or conceptual. (AKA "Entity" in some other context)

Object

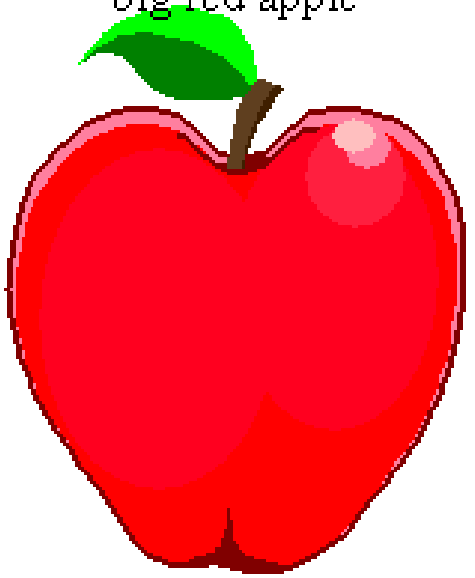
What is an Object ?

An Object is an abstraction of an Entity from some information processing point of view

hungry person



big red apple



Ochimizu, Higashida,"Object Modeling", Addison-Wesley Publishers Japan

Object

- An object has (1) states, (2) behaviors and an (3) identity.
- How an object reacts (or behaves) to your message depends on its current state. You need to know which object you interact with, so you need to address a specific object. Therefore, you need to be able to identify an object.

Object State

- An object has states. "State of the object is all the data which it currently encapsulates."
- It reacts to a message in different ways, depending on its state. To represent and store its states, an object usually can have many attributes (data member or instance variables). Some of these attribute is changeable, indicating a mutable state. The others are not changeable, indicating an immutable state.

Object Behavior

- An object has behaviors. "An object acts and reacts, in terms of state changes and message passing."
- This is the dynamic aspect of an object. It shall be able to "do things", so it possesses some kinds of behaviors. For the object to behave when called, it must be able to interpret the messages sent to it. In other words, it understands and acts on certain messages.

Object Identity

- An object has an identity. The object shall have a continuing existence.
- You need to identify it so that you can address it. An object's attribute values could change, but its identity shall remain the same.
- In many scenarios, you can use a name to identify an object. But the name is not the same thing as the object. Sometimes, you can assign different names to a single object. Considering the following example in Java code.

```
Car mycar = new Car();
//instantiating  a new object, storage is assigned to this new object in the system's memory space

Car yourcar = mycar;
//This LOC creates another reference variable that share sthe same value as that of "mycar". Now the
// previous created object has two names, "mycar" and "yourcar", they both refer to the same object.
```

  - This is where the CS definition of object kicks in. "In computer science, an object is a location in memory having a value and referenced by an identifier. An object can be a variable, function, or data structure." Mark the difference.

An Object Example

- We have an object called myClock. It has interfaces (for the callers) reportTime() and resetTimeTo(newTime: Time).

- It understands the messages like "reportTime" (calling the reportTime() interface), resetTimeTo(07:43) and resetTimeTo(12:30) (calling the resetTimeTo(newTime: Time) interface). (07:43 and 12:30 are arguments of the messages, and they are the Time objects)

- When reportTime is called, it grabs the current time and returns it to the caller. When resetTimeTo(newTime: Time) is called, it simply set the time to the value indicated by the newTime argument.

An Object Example

- How does it implement the functionalities for reportTime() and resetTimeTo(newTime:Time)? We don't know. We only know that it can deal with such messages.
- It reflects another major principle of CS/SE, information hiding.
- Information Hiding: Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Messages

- Suppose you know which object you want to send a message to. You need to know (1) what kind of behavior you want it to give it to you and (2) you need to provide some additional information for the object to consider when it reacts on your message.
- The first part we call a selector. The second part we call the arguments. A message may or may not have arguments.
- "reportTime" and "resetTimeTo" are selectors, "07:43", "12:30" are arguments. "reportTime" has no argument.
- Different languages have different ways to implement messages. The common practice is that the syntax defines the messages as methods, with method name and argument list. In the below is a java code example.

Messages

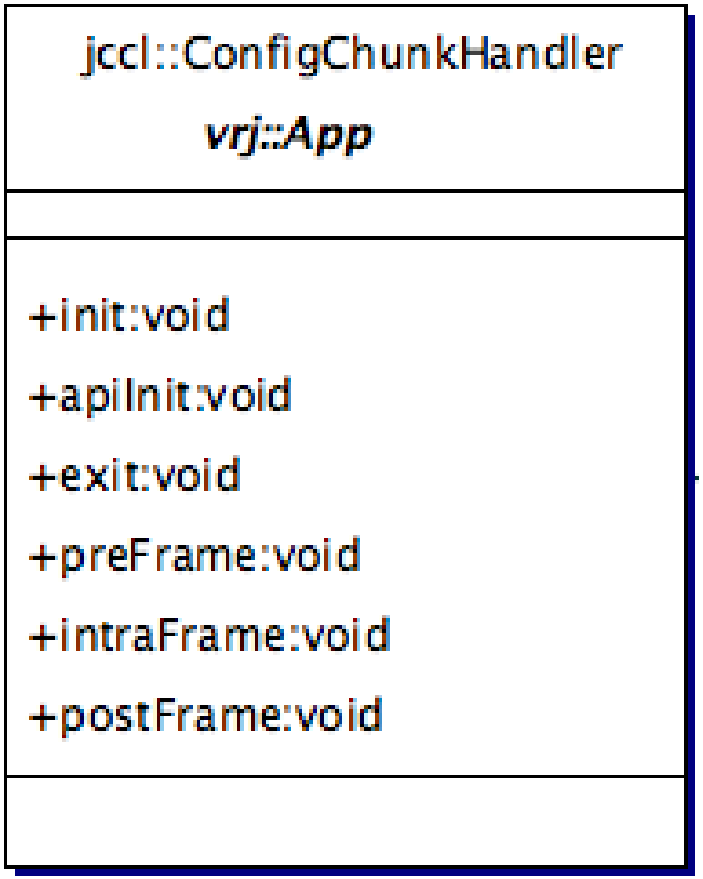
```
public class Bicycle {
    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;
    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
    // the Bicycle class has three methods
    public void setCadence(int newValue) {cadence = newValue;}
    public void setGear(int newValue) {gear = newValue;}
    public void applyBrake(int decrement) {speed -= decrement;}
}
```

Interfaces Revisited

- Every object has some interfaces. "An object's public interfaces define which messages it will accept regardless of where they come from."

- In OO, the specification about the interface is given in the comments or other accompanying documentations. (Not in the code, why? Abstraction)

- Sometimes, an object is typically able to understand some messages not declared by its public interfaces. (For example, a specific data model pertaining to a single object. The other parts (objects) of the system do not need to possess the knowledge about it. So the object doesn't publish any interfaces regarding its manipulation in its public interfaces.)


jccl::ConfigChunkHandler
vrj::App
+init:void
+apiInit:void
+exit:void
+preFrame:void
+intraFrame:void
+postFrame:void

Interfaces Revisited

- An object A's interfaces – A's public interfaces = A's private interfaces

- Why is it? It is also because of encapsulation. "You don't need to know more than enough about me: only I need to know it." Such knowledge is normally kept in the attributes. So the common practice is that we do not publish the attributes of an object in its public interfaces.

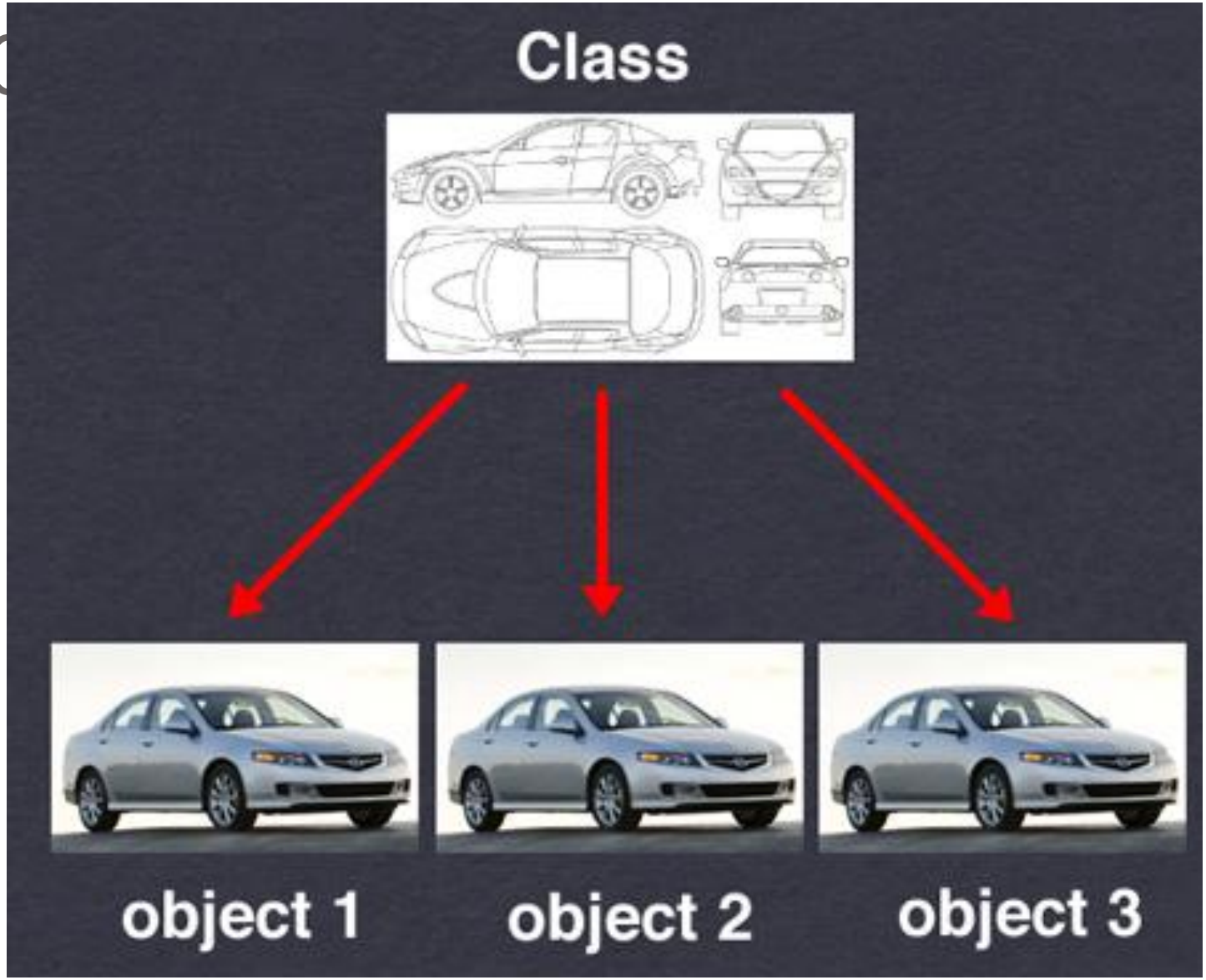
- Generally speaking, an object has two sets of interfaces, the public interfaces (understood by all objects) and the private interfaces (understood by itself, "+": public, "-": private).

Classes

- We can introduce another level of abstraction here by introducing the concept of class.
  - In object-oriented programming, a class is a construct that is used to create instances of itself – referred to as class instances, class objects, instance objects or simply objects.
  - A class defines constituent members which enable its instances to have state and behavior.
  - Data field members (member variables or instance variables) enable a class instance to maintain state. (They are actually the attributes)
- Other kinds of members, especially methods, enable the behavior of class instances. Classes define the type of their instances

Classes

- "In classed-based OO languages, every object belongs to a class, and the class of an object determines its interface."
- For example, myClock can be a Clock object. It can understand the resetTimeTo(mytime:Time) and reportTime() messages defined in the Clock class.
  - An object's attributes are also defined by its class, though their actual values can vary.
  - An object's methods (behaviors) are also defined by its class, though the actual behaviors effects are determined by the object's state (attribute values) and the method together.
  - A class can define public interfaces and private interfaces. (discussed in the last sub-section)
  - A class is an "object factory", it can create as many instances as itself as needed.


Class
object 1
object 2
object 3

Class as object

- The title may appeal a little strange. Class is a group of similar objects (with the same type of attributes and behaviors, or the same structures): then how can a class be an object?
- An object can indeed be a class, if you treat it as an "entity". (Remember that by definition an object can be any conceptual thing?) In fact "many OOP languages allow a class to behave as though it was an object in its own right, having attributes and understanding messages itself."
- For example, the class "Customer" might have an attribute "numberOfInstantces" which might be an integer incremented each time a new "Customer" object is created. It (the "Customer" class) might need this information for statistical concerns.
  - Static methods of classes in java.

Why Have Classes (why objects are not good enough)?

- Are objects (states, behaviors, identity) all we require for a good AD? NO.
- If we do not abstract objects (into classes), what happens ?
  - There is no need to store a copy of the code representing an object's behavior in every object, even though conceptually we think of every object as encapsulating its own code.
  - WRITE ONCE and USE EVERYWHERE!

Why Have Classes (why objects are not good enough)?

- Classes improves the readability of your program (system) and helps produces a good analysis of your system. (Class enhances the modularity of your system.)

How Does OO Help US Build a Good System?

- Modularity (Encapsulation, Abstraction, Low Coupling, High Cohesion, etc) are what we are looking for in a good system. How do we find it in OO?
- Individual objects are modules (states, behavior and identity satisfies encapsulation), only concerning objects would not bring abstraction. Get classes, so we can have both encapsulation and abstraction.
- Classes are loosely coupled and high cohesive modules.

How Does OO Help US Build a Good System?

- Another benefit of OO is that it brings natural representations of conceptual or real-world entities into your system. It improves the readability of your system and help produces a good system analysis.
  - "Real world entities (domain objects) change less frequently and less dramatically than the exact functionality that the user requires.
  - Functional vs OO decomposition of system

How Does OO Help US Build a Good System?

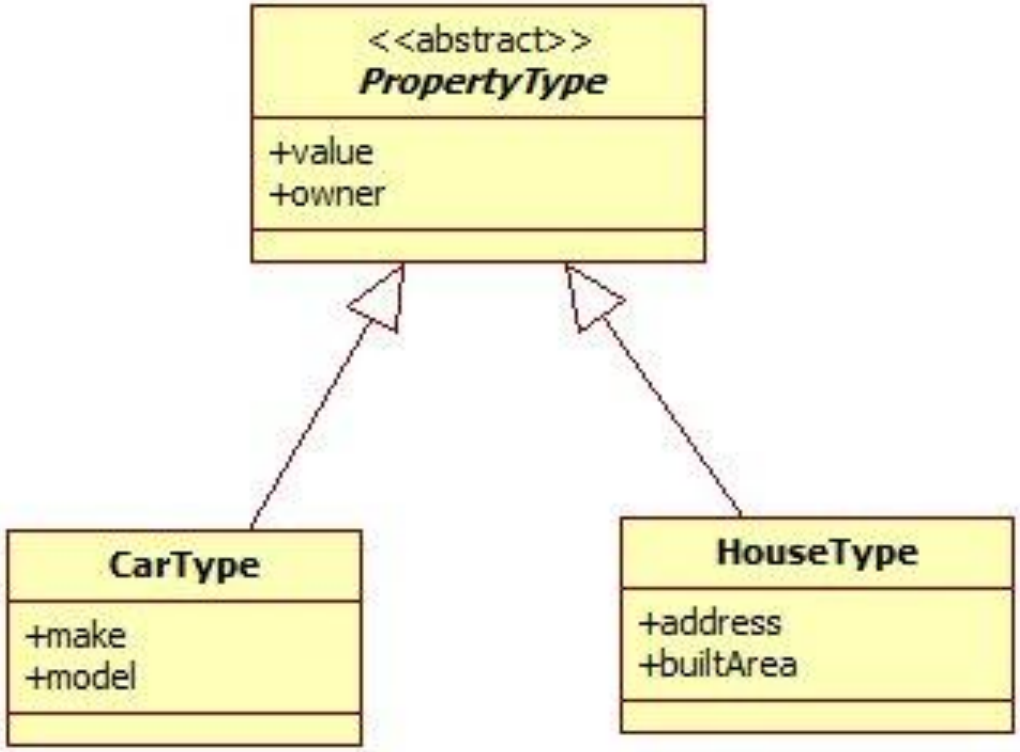
- Components using OO highly likely facilitates the construction of an architecture-oriented system (design) approach.
- In all,
  - The OO approach takes modularity, encapsulation and abstraction as fundamental
  - Components using OO facilitate the architecture-oriented approach.
  - And OOPLs make them (comparatively!) easy, so that there is reasonable likelihood that the obvious way to do something is also a good way to do it

Inheritance

- Consider an example.
- Lectures, Directors of Studies. A Director of Studies is a special kind of Lecturer: in addition to the normal duties of a lecturer, he is responsible for overseeing the progress of particular students.
- Suppose that in our system a *DirectorOfStudies* Object ought to understand the same messages as a *Lecturer* does, and in addition ought to be able to respond to the message *directees* by returning a collection of *Student* objects"
- Duplicate the code of Lecturer for DirectorOfStudies? Tradeoff?

Inheritance

- OOP allows us to define a new class DirectorOfStudies in terms of the old class Lecturer. "We simply specify that DirectorOfStudies is a subclass of Lecturer, and then type only what pertains to the extra attributes or operations of DirectorOfStudies" DirectorOfStudies can inherit all the attributes and methods from Lecturer.



Inheritance

- Inheritance: If class A is a sub-class for class B, then A inherits from B.
- We also say:
  - A is a specialization of B
  - A is more specialized than B
  - B is a superclass of A
  - B is a generalization of B

- A subclass is an extended, specialized version of its superclass. It includes the operations and attributes of the superclass, and possibly some more.

- More interestingly, we can override any method inherited. In this case, we can override some of *Lecture*'s methods in *DirectorOfStudies* to implement difference behaviors on receipt of some messages.

Polymorphism and Dynamic Binding

- Polymorphism and Dynamic Binding are used interchangeably in many OO literatures.
- The central idea is that: in OO, your object can have multiple types (classes) (polymorphism), when the object is called for actions, the system should determine which type it is (dynamic binding) before invoking its appropriate behaviors.

Polymorphism and Dynamic Binding

- Here is an example.
  - Suppose *Lecture* interfaces includes the operation canDo(duty:Duty), which is overridden in *DirectorOfStudies*.
  - Assume there is a collection of Lecturers, many of which are *DirectorOfStudies* objects.
  - When we execute the following code, what should happen when the for loop reaches an object o which is in fact in class *DirectorOfStudies*?

- *for each o in lecturers*
- *o.canDo(seminaOrganization)*

- Clearly, in this case the *DirectorOfStudies* code is executed.

Polymorphism and Dynamic Binding

- This is dynamic binding.
  - The same piece of syntax should cause more than one piece of code to be executed at different times. The message-send is dynamically bound to the appropriate code.

- Now here is polymorphism .
  - Polymorphism is a programming language feature that allows values of different data types to be handled using a uniform interface.