
STRUCTURE
CLASS
OOP
Object Oriented Programming
Lambton
College

Agenda

- Structure
- Class
- Object oriented programming
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism

Structure

- In database applications, structures are called records
- A collection of related data items, possibly of different types
- A structure type in Swift is called struct
- A struct is a heterogenous collection and it can be composed of data of different types
- A single struct would store the data for one object
- Structures in Swift can also contain functions

Structure

- Individual components of a struct type called members (or field)
- Functions inside a struct is called method
- Members can be of different types (simple, array or struct)
- We can have an array of structures in our program
- To create a new instance of the structure in swift, we need to call its initializer
- Structures in Swift receive a default member wise initializer
- Structures in Swift are value types

Class

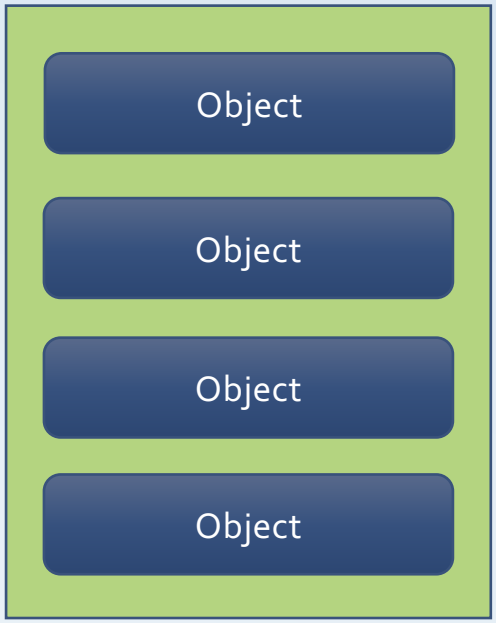
- The mechanism that allows you to combine data and the function in a single unit is called a class
- Like structures a class is the collection of related data and function under a single name
- Once a class is defined, you can declare variables of that type
- Like structures a class variable is called object or instance i.e. a class would be the data type, and an object would be the variable
- Classes are declared using the keyword class
- An instance of a class can be inherited and we can have several instances of that class
- The behavior of a method of an instance may be different in the inherited instances i.e. polymorphism

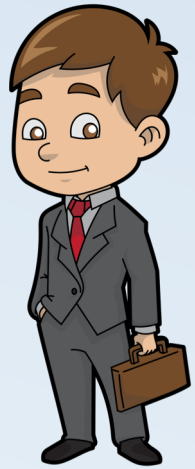
Class

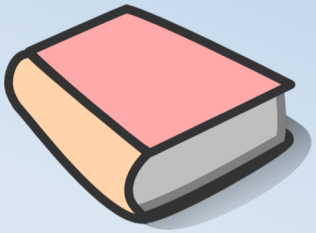
- A class is a definition of an object. It's a type just like Int
- A class is a type, and an object of this class is just a variable
- Individual components of a class type called members (or field)
- Functions inside a class is called method
- We can have an array of objects in our program
- To create a new instance of the class in swift, we need to call its initializer
- classes in Swift don't receive a default member wise initializer
- classes in Swift are reference types

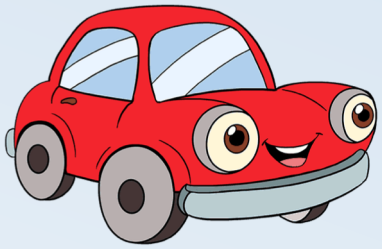
- OOP

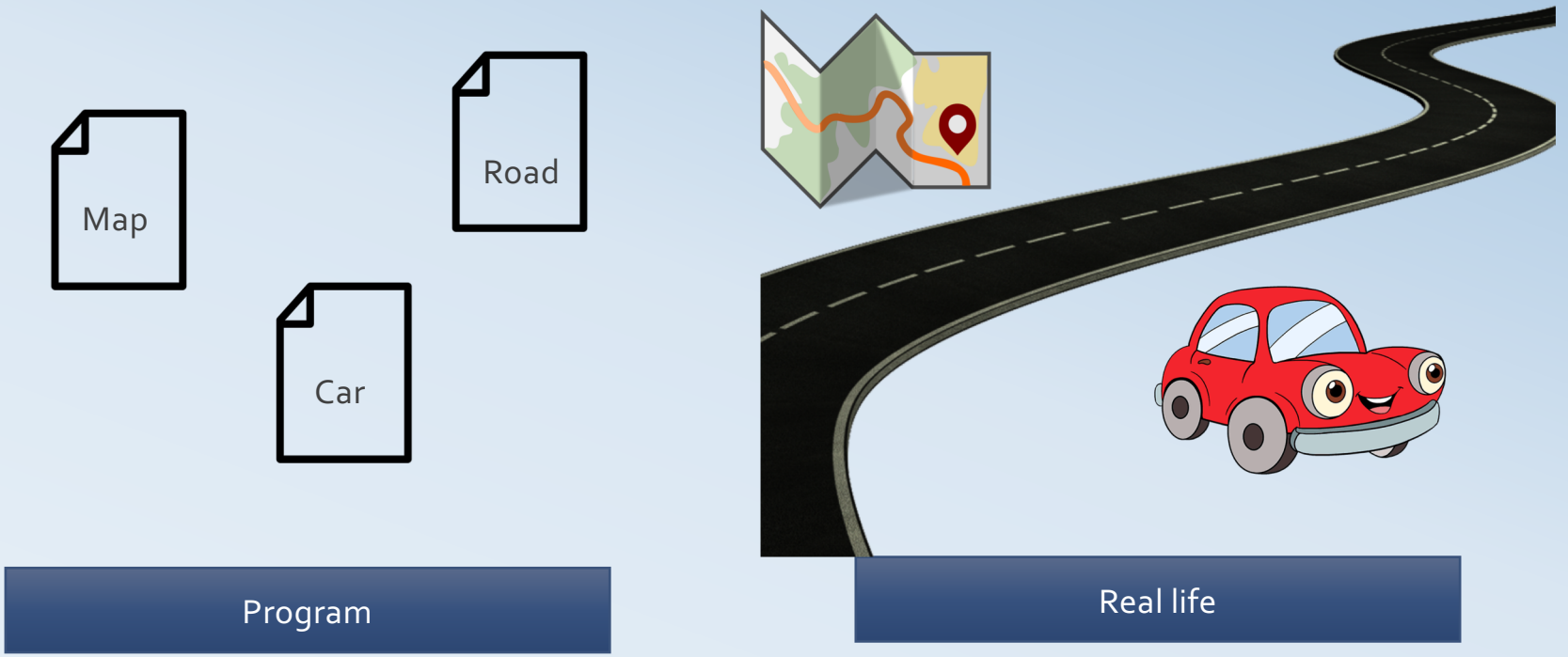
- OOP stands for Object Oriented Programming





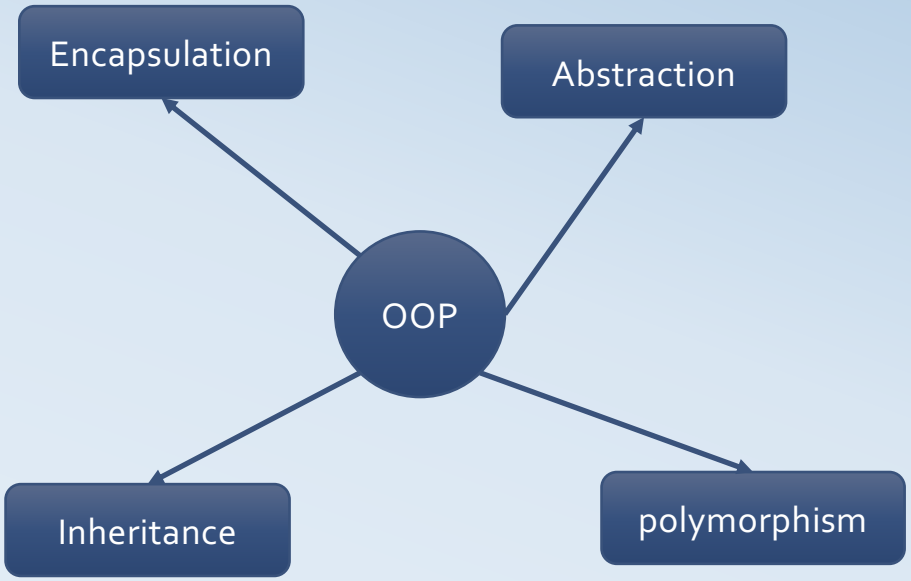






OOP
when you are coding, you want to solve a real world problem

- Class
- Object
- Abstraction
- Encapsulation
- Inheritance
- polymorphism



Concept of OOP

Class

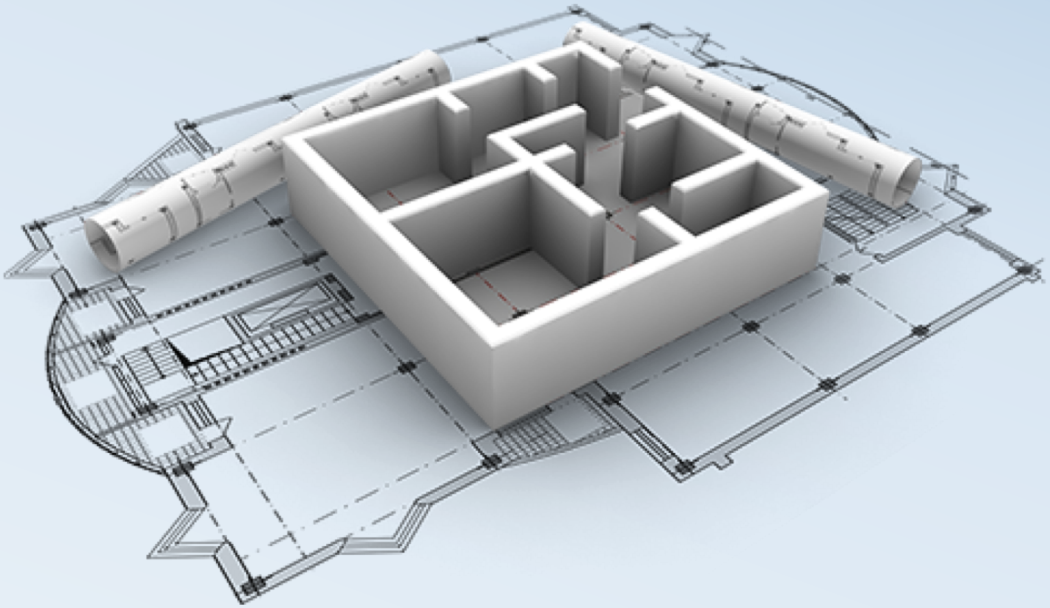
- A **class** is a data type that allows programmers to create objects. A class provides a definition for an object, describing an object's attributes (data) and methods (operations)

Object

- An object is an *instance* of a class. With one class, you can have as many objects as required
- This is analogous to a variable and a data type, the class is the data type and the object is the variable

OOP

- It's a blueprint, the definition and the description



Class

- The object is created from the class
- One class can create multiple objects

Object

Class and Objects

- **Class** – Provides a way to create new objects based on a "meta-definition" of an object (Example: The Person **class**)
- **Object** – Unique programming entity that has methods, has attributes and can react to events
- **Method** – Things which an object can do; the "verbs" of objects. In code, usually can be identified by an "action" word
- **Attribute** – Things which describe an object; the "adjectives" of objects. In code, usually can be identified by a "descriptive" word
- **Initializer** – Special methods used to create new instances of a class (Example: A Honda Civic is an **instance** of the automobile **class**.)

Object Oriented Programming

- What does a class define?

ATTRIBUTES	BEHAVOIR
name height weight gender age …	walk run jump speak sleep …

- Attributes in programming term is called properties or fields and behavior in the same way is called methods.
- The class is describing these things in abstract
- **Definition**: The act of representing essential features without including the background details or explanations

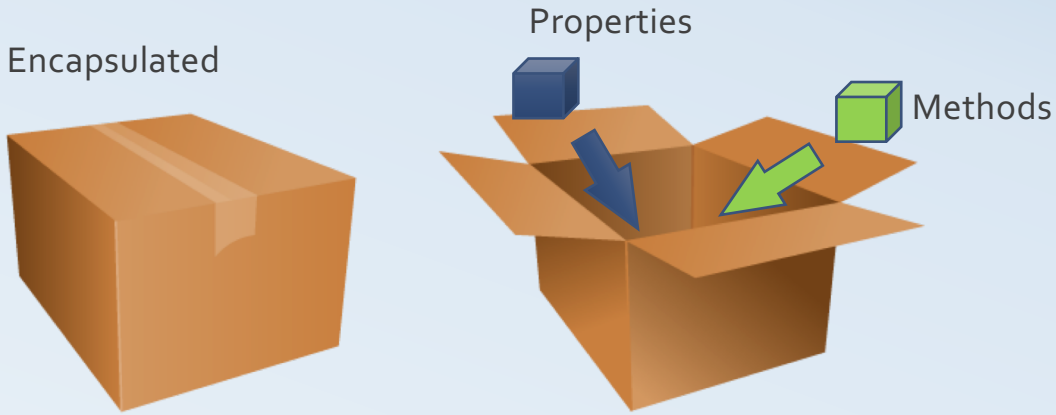
Abstraction

- Incorporation into a class of data & operations in one package
- Data can only be accessed through that package
- "Information Hiding"

Encapsulation

Definition: the wrapping up the properties and methods into a single unit called class

- A language mechanism for restricting access to some of the object's components
- Hiding the implementation details and providing restrictive access leads to the concept of abstract data type

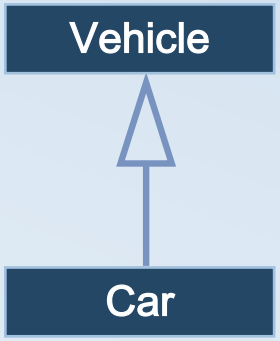


Encapsulation

Inheritance

- Another fundamental object-oriented technique is inheritance, used to organize and create reusable classes
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class*, or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class
- New methods and attributes can be created in the new class, but don't affect the parent class's definition
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance

Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class



Inheritance should create an is-a relationship, meaning the child is a more specific version of the parent

Inheritance

- Initializers are not inherited by default
- Yet we often want to use the parent's initializer to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's initializer
- A child's initializer is responsible for calling the parent's initializer
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

The super reference

- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Swift supports *single inheritance*, meaning that a derived class can have only one parent class
- Swift does not support multiple inheritance
- In most cases, the use of protocols gives us aspects of multiple inheritance

Multiple Inheritance

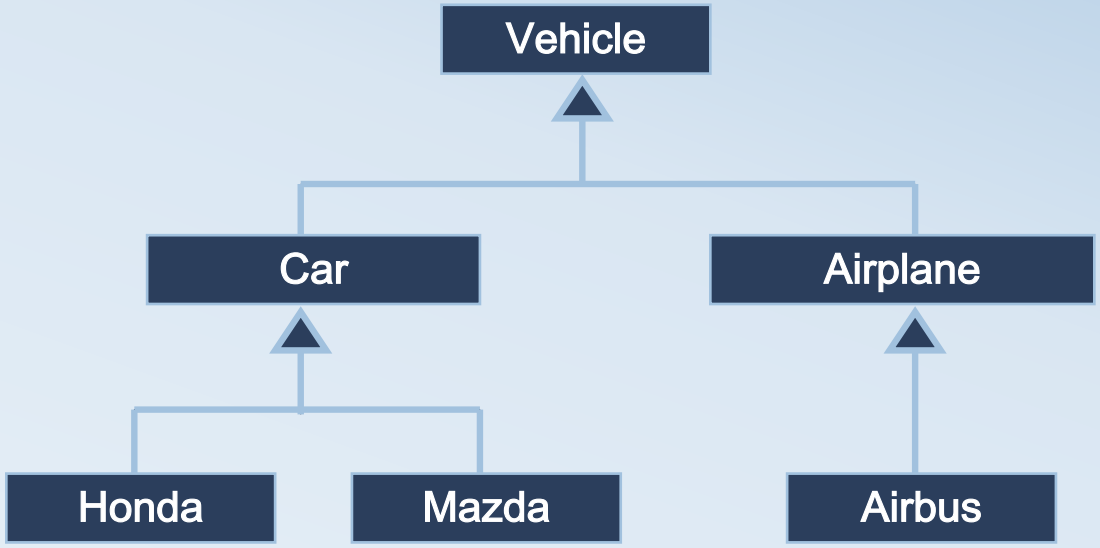
- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
- A parent method can be invoked explicitly using the `super` reference
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overriding Methods

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Overloading vs. Overriding

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

Class Hierarchies

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- There are no abstract classes in Swift
- You can achieve the same behaviour with protocols and protocol extensions

Abstract Classes

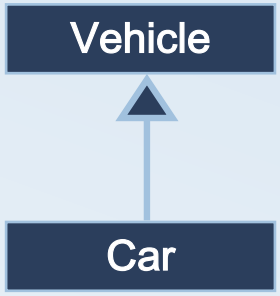
- A reference can be *polymorphic*, which can be defined as "having many forms"

```
obj.doIt()
```

- This line of code might execute different methods at different times if the object that `obj` points to changes
- Careful use of polymorphic references can lead to elegant, robust software designs
- Polymorphism can be accomplished using inheritance or using protocols

Polymorphism

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the *Vehicle* class is used to derive a child class called *Car*, then a *Vehicle* reference could be used to point to a *Car* object




```
var myCar : Vehicle
myCar = Car()
```

References and Inheritance

- Assigning a descendant object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment (upcasting)
- Assigning an ancestor object to a descendant reference can be done also, but it is considered to be a narrowing conversion and must be done with a caution (downcasting)
- The widening conversion is the most useful

References and Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Vehicle` class has a method called `performance`, and the `Car` class overrides it
- Now consider the following invocation:

```
myCar.performance()
```

- If `myCar` refers to a `Vehicle` object, it invokes the `Vehicle` version of `performance`; if it refers to a `Car` object, it invokes the `Car` version

Polymorphism via Inheritance

- Inheritance can be applied to protocols as well as classes
- One protocol can be derived from another protocol
- The child protocol inherits all abstract methods of the parent
- A class implementing the child protocol must define all methods from both the ancestor and child protocols
- Note that class hierarchies and protocol hierarchies are distinct (they do not overlap)

Protocol hierarchies

Polymorphism via Interfaces

Suppose the following code:

- The classes Airplane and Bird must implement the adopted Flyable protocol
- The method fly has a polymorphic behavior depending on the class that implements the Flyable protocol and define the method fly()

```
protocol Flyable {
  func fly() -> String
}

class Airplane: Flyable {

  func fly() -> String {
    return ("Airplane can fly")
  }

}

class Bird: Flyable {

  func fly() -> String {
    return ("Bird can fly too")
  }

}
```

Which of the following is not part of OOP?

- ❑ Polymorphism
- ❑ Abstraction
- ❑ Data encapsulation
- ☑ Multitasking
- ❑ Inheritance

Quiz

Thank you for your Attention