Object-Oriented Programming

Introduction

What

: ★☆☆☆ Can describe OOP at a higher level

Object-Oriented Programming (OOP) is a *programming paradigm*. A programming paradigm guides programmers to analyze programming problems, and structure programming solutions, in a specific way.

> Programming languages have traditionally divided the world into two parts—data and operations on data. Data is static and immutable, except as the operations may change it. The procedures and functions that operate on data have no lasting state of their own; they're useful only in their ability to affect data.
>
> This division is, of course, grounded in the way computers work, so it's not one that you can easily ignore or push aside. Like the equally pervasive distinctions between matter and energy and between nouns and verbs, it forms the background against which you work. At some point, all programmers—even object-oriented programmers—must lay out the data structures that their programs will use and define the functions that will act on the data.
>
> With a procedural programming language like C, that's about all there is to it. The language may offer various kinds of support for organizing data and functions, but it won't divide the world any differently. Functions and data structures are the basic elements of design.
>
> Object-oriented programming doesn't so much dispute this view of the world as restructure it at a higher level. It groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.
>
> -- Object-Oriented Programming with Objective-C, Apple

Some other examples of programming paradigms are:

Paradigm	Programming Languages
Procedural Programming paradigm	C
Functional Programming paradigm	F#, Haskell, Scala
Logic Programming paradigm	Prolog

Some programming languages support multiple paradigms.

Java is primarily an OOP language but it supports limited forms of functional programming and it can be used to (although not recommended) write procedural code. e.g. se-edu/addressbook-level1

JavaScript and Python support functional, procedural, and OOP programming.

Exercises

Objects

What

🎓: ★★☆☆ 🏆 Can describe how OOP relates to the real world

> Every object has both state (data) and behavior (operations on data). In that, they're not much different from ordinary physical objects. It's easy to see how a mechanical device, such as a pocket watch or a piano, embodies both state and behavior. But almost anything that's designed to do a job does, too. Even simple things with no moving parts such as an ordinary bottle combine state (how full the bottle is, whether or not it's open, how warm its contents are) with behavior (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand high or low temperatures).
>
> It's this resemblance to real things that gives objects much of their power and appeal. They can not only model components of real systems, but equally as well fulfill assigned roles as components in software systems.
>
> -- Object-Oriented Programming with Objective-C, Apple

Object Oriented Programming (OOP) views the world as *a network of interacting objects*.

📦 A real world scenario viewed as a network of interacting objects:

You are asked to find out the average age of a group of people Adam, Beth, Charlie, and Daisy. You take a piece of paper and pen, go to each person, ask for their age, and note it down. After collecting the age of all four, you enter it into a calculator to find the total. And then, use the same calculator to divide the total by four, to get the average age. This can be viewed as the objects `You`, `Pen`, `Paper`, `Calculator`, `Adam`, `Beth`, `Charlie`, and `Daisy` interacting to accomplish the end result of calculating the average age of the four persons. These objects can be considered as connected in a certain network of certain structure.

OOP solutions try to create a similar object network inside the computer's memory – a sort of virtual simulation of the corresponding real world scenario – **so that a similar result can be achieved programmatically.**

OOP does not demand that the virtual world object network follow the real world *exactly*.

📦 Our previous example can be tweaked a bit as follows:

- Use an object called `Main` to represent your role in the scenario.
- As there is no physical writing involved, you can replace the `Pen` and `Paper` with an object called `AgeList` that is able to keep a list of ages.

Every object has both state (data) and behavior (operations on data).

Object	Real World?	Virtual World?	Example of State (i.e. Data)	Examples of Behavior (i.e. Operations)
Adam	✔	✔	Name, Date of Birth	Calculate age based on birthday
Pen	✔	-	Ink color, Amount of ink remaining	Write
AgeList	-	✔	Recorded ages	Give the number of entries, Accept an entry to record
Calculator	✔	✔	Numbers already entered	Calculate the sum, divide
You/Main	✔	✔	Average age, Sum of ages	Use other objects to calculate

Every object has an *interface* and an *implementation*.

Every real world object has:

- an interface through which other objects can interact with it
- an implementation that supports the interface but may not be accessible to the other object

📦 The interface and implementation of some real-world objects in our example:

- Calculator: the buttons and the display are part of the interface; circuits are part of the implementation.
- Adam: In the context of our 'calculate average age' example, the interface of Adam consists of requests that Adam will respond to, e.g. "Give age to the nearest year, as at Jan 1st of this year" "State your name"; the implementation includes the mental calculation Adam uses to calculate the age which is not visible to other objects.

Similarly, every object in the virtual world has an interface and an implementation.

📦 The interface and implementation of some virtual-world objects in our example:

- `Adam`: the interface might have a method `getAge(Date asAt)`; the implementation of that method is not visible to other objects.

Objects interact by sending messages. Both real world and virtual world object interactions can be viewed as objects sending messages to each other. The message can result in the sender object receiving a response and/or the receiver object's state being changed. Furthermore, the result can vary based on which object received the message, even if the message is identical (see rows 1 and 2 in the example below).

Examples:

World	Sender	Receiver	Message	Response	State Change
Real	You	Adam	"What is your name?"	"Adam"	-
Real	as above	Beth	as above	"Beth"	-
Real	You	Pen	Put nib on paper and apply pressure	Makes a mark on your paper	Ink level goes down
Virtual	Main	Calculator (current total is 50)	add(int i): int i = 23	73	total = total + 23

Exercises

Objects as Abstractions

🎓: ★★☆☆ 🏆 Can explain the abstraction aspect of OOP

The concept of ***Objects*** **in OOP is an abstraction mechanism because it allows us to abstract away the lower level details and work with bigger granularity entities** i.e. ignore details of data formats and the method implementation details and work at the level of objects.

📦 You can deal with a `Person` object that represents the person Adam and query the object for Adam's age instead of dealing with details such as Adam's date of birth (DoB), in what format the DoB is stored, the algorithm used to calculate the age from the DoB, etc.

Encapsulation Of Objects

🎓: ★★☆☆ 🏆 Can explain the encapsulation aspect of OOP

> Encapsulation protects an implementation from unintended actions and from inadvertent access.
>
> -- Object-Oriented Programming with Objective-C, Apple

An object is an *encapsulation* of some data and related behavior in terms of two aspects:

1. The *packaging* aspect: An object packages data and related behavior together into one self-contained unit.

2. The *information hiding* aspect: The data in an object is hidden from the outside world and are only accessible using the object's interface.

Exercises

Classes

What

🎓: ★☆☆☆ 🏆 Can explain the relationship between classes and objects

Writing an OOP program is essentially writing instructions that the computer will use to,

1. **create the virtual world of the object network, and**
2. **provide it the inputs to produce the outcome you want.**

A *class* contains instructions for creating a specific kind of objects. It turns out sometimes multiple objects keep the same type of data and have the same behavior because they are of the *same kind*. Instructions for creating a 'kind' (or 'class') of objects can be done once and those same instructions can be used to *instantiate* objects of that kind. You call such instructions a *Class*.

Classes and objects in an example scenario

Consider the example of writing an OOP program to calculate the average age of Adam, Beth, Charlie, and Daisy.

Instructions for creating objects `Adam`, `Beth`, `Charlie`, and `Daisy` will be very similar because they are all of the same kind: they all represent 'persons' with the same interface, the same kind of data (i.e. `name`, `dateOfBirth`, etc.), and the same kind of behavior (i.e. `getAge(Date)`, `getName()`, etc.). Therefore, you can have a class called `Person` containing instructions on how to create `Person` objects and use that class to instantiate objects `Adam`, `Beth`, `Charlie`, and `Daisy`.

Similarly, you need classes `AgeList`, `Calculator`, and `Main` classes to instantiate one each of `AgeList`, `Calculator`, and `Main` objects.

Class	Objects
`Person`	objects representing Adam, Beth, Charlie, Daisy
`AgeList`	an object to represent the age list

Class	Objects
`Calculator`	an object to do the calculations
`Main`	an object to represent you (i.e., the one who manages the whole operation)

Exercises

Identify Classes and Objects

Classes for CityConnect app

Class Level Members

: ★★☆☆ Can explain class-level members

While all objects of a class have the same attributes, each object has its own copy of the attribute value.

All `Person` objects have the `name` attribute but the value of that attribute varies between `Person` objects.

However, some attributes are not suitable to be maintained by individual objects. Instead, they should be maintained centrally, shared by all objects of the class. They are like 'global variables' but attached to a specific class. Such **variables whose value is shared by all instances of a class are called *class-level attributes***.

The attribute `totalPersons` should be maintained centrally and shared by all `Person` objects rather than copied at each `Person` object.

Similarly, when a normal method is being called, a message is being sent to the receiving object and the result may depend on the receiving object.

Sending the `getName()` message to the `Adam` object results in the response `"Adam"` while sending the same message to the `Beth` object results in the response `"Beth"`.

However, there can be methods related to a specific class but not suitable for sending messages to a specific object of that class. Such **methods that are called using the class instead of a specific instance are called *class-level methods***.

The method `getTotalPersons()` is not suitable to send to a specific `Person` object because a specific object of the `Person` class should not have to know about the total number of `Person` objects.

Class-level attributes and methods are collectively called *class-level members* (also called *static members* sometimes because some programming languages use the keyword `static` to identify class-level members). **They are to be accessed using the class name rather than an instance of the class**.

Exercises

Enumerations

: ★★☆☆ Can explain the meaning of enumerations

An *Enumeration* is a fixed set of values that can be considered as a data type. An enumeration is often useful when using a regular data type such as `int` or `String` would allow invalid values to be assigned to a variable.

Suppose you want a variable called `priority` to store the priority of something. There are only three priority levels: high, medium, and low. You can declare the variable `priority` as of type `int` and use only values `2`, `1`, and `0` to indicate the three priority levels. However, this opens the possibility of an invalid value such as `9` being assigned to it. But if you define an enumeration type called `Priority` that has three values `HIGH`, `MEDIUM` and `LOW` only, a variable of type `Priority` will never be assigned an invalid value because the compiler is able to catch such an error.

`Priority`: `HIGH`, `MEDIUM`, `LOW`

Associations

What

★☆☆☆ Can explain associations

Objects in an OO solution need to be connected to each other to form a network so that they can interact with each other. Such **connections between objects are called *associations*.**

Suppose an OOP program for managing a learning management system creates an object structure to represent the related objects. In that object structure you can expect to have associations between a `Course` object that represents a specific course and `Student` objects that represent students taking that course.

Associations in an object structure can change over time.

To continue the previous example, the associations between a `Course` object and `Student` objects can change as students enroll in the module or drop the module over time.

Associations among objects can be generalized as associations between the corresponding classes too.

In our example, as some `Course` objects can have associations with some `Student` objects, you can view it as an association between the `Course` class and the `Student` class.

Implementing associations

You use instance level variables to implement associations.

Navigability

★★☆☆ Can explain the meaning of navigability

When two classes are linked by an association, it does not necessarily mean the two objects taking part in an instance of the association *knows about* (i.e., has a reference to) each other. **The concept of which object in the association knows about the other object is called *navigability*.**

Navigability can be *unidirectional* or *bidirectional*. Suppose there is an association between the classes `Box` and `Rope`, and the `Box` object `b` and the `Rope` object `r` is taking part in one instance of that association.

- **Unidirectional**: If the navigability is from `Box` to `Rope`, `b` will have a reference to `r` but `r` will not have a reference to `b`. Similarly, if the navigability is in the other direction, `r` will have a reference to `b` but `b` will not have a reference to `r`.
- **Bidirectional**: `b` will have a reference to `r` and `r` will have a reference to `b` i.e., the two objects will be pointing to each other for the same single instance of the association.

Note that two unidirectional associations in opposite directions do not add up to a single bidirectional association.

In the code below, there is a bidirectional association between the `Person` class and the `Cat` class i.e., if `Person` `p` is the owner of the `Cat` `c`, `p` it will result in `p` and `c` having references to each other.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person owner;
    //...
}
```

The code below has two unidirectional associations between the `Person` class and the `Cat` class (in opposite directions) because the breeder is not necessarily the same person keeping the cat as a pet i.e., there are two separate associations here, which rules out it being a bidirectional association.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person breeder;
    //...
}
```

Multiplicity

★★☆☆ Can explain the meaning of multiplicity

***Multiplicity* is the aspect of an OOP solution that dictates how many objects take part in each association.**

The multiplicity of the association between `Course` objects and `Student` objects tells you how many `Course` objects can be associated with one `Student` object and vice versa.

Implementing multiplicity

A normal instance-level variable gives us a `0..1` multiplicity (also called *optional associations*) because a variable can hold a reference to a single object or `null`.

In the code below, the `Logic` class has a variable that can hold `0..1` i.e., zero or one `Minefield` objects.

```
class Logic {
    Minefield minefield;
    // ...
}

class Minefield {
    //...
}
```

A variable can be used to implement a `1` multiplicity too (also called *compulsory associations*).

In the code below, the `Logic` class will always have a `ConfigGenerator` object, provided the variable is not set to `null` at some point.

```
class Logic {
    ConfigGenerator cg = new ConfigGenerator();
    ...
}
```

Bidirectional associations require matching variables in both classes.

In the code below, the `Foo` class has a variable to hold a `Bar` object and vice versa i.e., each object can have an association with an object of the other type.

```
class Foo {
    Bar bar;
    //...
}

class Bar {
    Foo foo;
    //...
}

```

To implement other multiplicities, choose a suitable data structure such as Arrays, ArrayLists, HashMaps, Sets, etc.

This code uses a two-dimensional array to implement a 1-to-many association from the `Minefield` to `Cell`.

```
class Minefield {
    Cell[][] cell;
    ...
}
```

Dependencies

★★★☆ Can explain dependencies among classes

In the context of OOP associations, **a *dependency* is a need for one class to depend on another without having a direct association in the same direction.** Reason for the exclusion: If there is an association from class `Foo` to class `Bar` (i.e., navigable from `Foo` to `Bar`), that means `Foo` is *obviously* dependent on `Bar` and hence there is no point in mentioning *dependency* specifically. In other words, we are only concerned about *non-obvious* dependencies here. One cause of such dependencies is interactions between objects that do not have a long-term link between them.

A `Course` class can have a dependency on a `Registrar` class because the `Course` class needs to refer to the `Registrar` class to obtain the the maximum number of students it can support (e.g., `Registrar.MAX_COURSE_CAPACITY`).

In the code below, `Foo` has a dependency on `Bar` but it is not an association because it is only a transient interaction and there is no long term relationship between a `Foo` object and a `Bar` object. i.e. the `Foo` object does not keep the `Bar` object it receives as a parameter.

```
class Foo {

    int calculate(Bar bar) {
        return bar.getValue();
    }
}

class Bar {
    int value;

    int getValue() {
        return value;
    }
}
```

Composition

★★★☆ Can explain the meaning of composition

A composition is an association that represents a strong *whole-part* relationship. When the *whole* is destroyed, *parts* are destroyed too i.e., the *part* should not exist without being attached to a *whole*.

A `Board` (used for playing board games) consists of `Square` objects.

Composition also implies that there cannot be cyclical links.

The 'sub-folder' association between `Folder` objects is a composition type association. That means if the `Folder` object `foo` is a sub-folder of `Folder` object `bar`, `bar` cannot be a sub-folder of `foo`.

Whether a relationship is a composition can depend on the context.

Is the relationship between `Email` and `EmailSubject` composition? That is, is the email subject *part* of an email to the extent that an email subject cannot exist without an email?

- When modeling an application that sends emails, the answer is 'yes'.
- When modeling an application that gather analytics about email traffic, the answer may be 'no' (e.g., the application might collect just the email subjects for text analysis).

A common use of composition is when parts of a big class are carved out as smaller classes for the ease of managing the internal design. In such cases, the classes extracted out still act as *parts* of the bigger class and the outside world has no business knowing about them.

Cascading deletion alone is not sufficient for composition. Suppose there is a design in which `Person` objects are attached to `Task` objects and the former get deleted whenever the latter is deleted. This fact alone does not mean there is a composition relationship between the two classes. For it to be composition, a `Person` must be an integral *part* of a `Task` in the context of that association, at the concept level (not simply at implementation level).

Identifying and keeping track of composition relationships in the design has benefits such as helping to maintain the data integrity of the system. For example, when you know that a certain relationship is a composition, you can take extra care in your implementation to ensure that when the *whole* object is deleted, all its *parts* are deleted too.

Implementing composition

Composition is implemented using a normal variable. If correctly implemented, the 'part' object will be deleted when the 'whole' object is deleted. Ideally, the 'part' object may not even be visible to clients of the 'whole' object.

In this code, the `Email` has a composition type relationship with the `Subject` class, in the sense that the subject is part of the email.

```
class Email {
    private Subject subject;
  ...
}
```

Aggregation

★★★☆ Can explain the meaning of aggregations

***Aggregation* represents a *container-contained* relationship.** It is a weaker relationship than composition.

`SportsClub` can act as a *container* for `Person` objects who are members of the club. `Person` objects can survive without a `SportsClub` object.

Implementing aggregation

Implementation is similar to that of composition except the *containee* object can exist even after the *container* object is deleted.

In the code below, there is an aggregation association between the `Team` class and the `Person` class in that a `Team` contains a `Person` object who is the leader of the team.

```
class Team {
    Person leader;
    ...
    void setLeader(Person p) {
        leader = p;
    }
}
```

Association Classes

★★★☆ Can explain the meaning of association classes

An *association class* represents additional information about an association. It is a normal class but plays a special role from a design point of view.

A `Man` class and a `Woman` class are linked with a 'married to' association and there is a need to store the date of marriage. However, that data is related to the association rather than specifically owned by either the `Man` object or the `Woman` object. In such situations, an additional association class can be introduced, e.g. a `Marriage` class, to store such information.

Implementing association classes

There is no special way to implement an association class. It can be implemented as a normal class that has variables to represent the endpoint of the association it represents.

In the code below, the `Transaction` class is an association class that represents a transaction between a `Person` who is the seller and another `Person` who is the buyer.

```
class Transaction {

    //all fields are compulsory
    Person seller;
    Person buyer;
    Date date;
    String receiptNumber;

    Transaction(Person seller, Person buyer, Date date, String receiptNumber) {
        //set fields
    }
}
```

Exercises

Inheritance

What

: ★☆☆☆ Can explain the meaning of inheritance

The OOP concept *Inheritance* allows you to define a new class based on an existing class.

For example, you can use inheritance to define an `EvaluationReport` class based on an existing `Report` class so that the `EvaluationReport` class does not have to duplicate data/behaviors that are already implemented in the `Report` class. The `EvaluationReport` can inherit the `wordCount` attribute and the `print()` method from the *base class* `Report`.

- Other names for Base class: *Parent* class, *Superclass*
- Other names for Derived class: *Child* class, *Subclass*, *Extended* class

A superclass is said to be *more general* than the subclass. Conversely, a subclass is said to be more *specialized* than the superclass.

Applying inheritance on a group of similar classes can result in the common parts among classes being extracted into more general classes.

`Man` and `Woman` behave the same way for certain things. However, the two classes cannot be simply replaced with a more general class `Person` because of the need to distinguish between `Man` and `Woman` for certain other things. A solution is to add the `Person` class as a superclass (to contain the code common to men and women) and let `Man` and `Woman` inherit from `Person` class.

Inheritance implies the derived class can be considered as a *sub-type* of the base class (and the base class is a *super-type* of the derived class), resulting in an *is a* relationship.

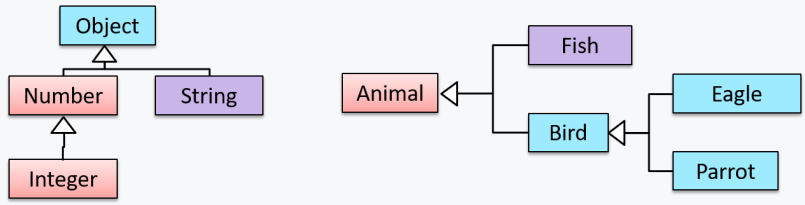
Inheritance does not necessarily mean a sub-type relationship exists. However, the two often go hand-in-hand. For simplicity, at this point let us assume inheritance implies a sub-type relationship.

To continue the previous example,

- `Woman` *is a* `Person`
- `Man` *is a* `Person`

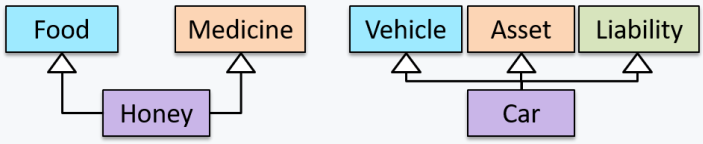
Inheritance relationships through a chain of classes can result in inheritance *hierarchies* (aka inheritance *trees*).

Two inheritance hierarchies/trees are given below. Note that the triangle points to the parent class. Observe how the `Parrot` *is a* `Bird` as well as it *is an* `Animal`.



***Multiple Inheritance* is when a class inherits *directly* from multiple classes.** Multiple inheritance among classes is allowed in some languages (e.g., Python, C++) but not in other languages (e.g., Java, C#).

The `Honey` class inherits from the `Food` class *and* the `Medicine` class because honey can be consumed as a food as well as a medicine (in some oriental medicine practices). Similarly, a `Car` is a `Vehicle`, an `Asset` and a `Liability`.



Exercises

Overriding

★☆☆☆ Can explain method overriding

Method *overriding* is when a sub-class changes the behavior inherited from the parent class by re-implementing the method. Overridden methods have the same name, same type signature, and same return type.

Consider the following case of `EvaluationReport` class inheriting the `Report` class:

`Report` methods	`EvaluationReport` methods	Overrides?
`print()`	`print()`	Yes
`write(String)`	`write(String)`	Yes
`read():String`	`read(int):String`	No. Reason: the two methods have different signatures; This is a case of *overloading* (rather than overriding).

Exercises

Overloading

: ★★☆☆ Can explain method overloading

Method overloading is when there are multiple methods with the same name but different type signatures. Overloading is used to indicate that multiple operations do similar things but take different parameters.

Type signature: The *type signature* of an operation is the type sequence of the parameters. The return type and parameter names are not part of the type signature. However, the parameter order is significant.

Example:

Method	Type Signature
`int add(int X, int Y)`	`(int, int)`
`void add(int A, int B)`	`(int, int)`
`void m(int X, double Y)`	`(int, double)`
`void m(double X, int Y)`	`(double, int)`

In the case below, the `calculate` method is overloaded because the two methods have the same name but different type signatures `(String)` and `(int)`.

- `calculate(String): void`
- `calculate(int): void`

Interfaces

🎓: ★★★☆ 🏆 Can explain interfaces

An *interface* is a behavior specification i.e. a collection of method specifications. If a class implements the interface, it means the class is able to support the behaviors specified by the said interface.

> There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts. --Oracle Docs on Java

📦 Suppose `SalariedStaff` is an interface that contains two methods `setSalary(int)` and `getSalary()`. `AcademicStaff` can declare itself as *implementing* the `SalariedStaff` interface, which means the `AcademicStaff` class must implement all the methods specified by the `SalariedStaff` interface i.e., `setSalary(int)` and `getSalary()`.

A class implementing an interface results in an *is-a* relationship, just like in class inheritance.

📦 In the example above, `AcademicStaff` *is a* `SalariedStaff`. An `AcademicStaff` object can be used anywhere a `SalariedStaff` object is expected e.g. `SalariedStaff ss = new AcademicStaff()`.

Abstract Classes

🎓: ★★★☆ 🏆 Can implement abstract classes

🌐 **Abstract class**: A class declared as an *abstract class* cannot be instantiated, but it can be subclassed.

You can declare a class as *abstract* when a class is merely a representation of commonalities among its subclasses in which case it does not make sense to instantiate objects of that class.

📦 The `Animal` class that exists as a generalization of its subclasses `Cat`, `Dog`, `Horse`, `Tiger` etc. can be declared as abstract because it does not make sense to instantiate an `Animal` object.

🌐 **Abstract method**: An *abstract method* is a method signature without a method implementation.

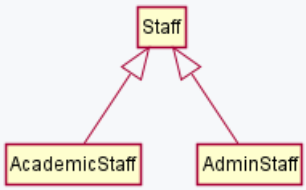
📦 The `move` method of the `Animal` class is likely to be an abstract method as it is not possible to implement a `move` method at the `Animal` class level to fit all subclasses because each animal type can move in a different way.

A class that has an abstract method becomes an abstract class because the class definition is incomplete (due to the missing method body) and it is not possible to create objects using an incomplete class definition.

Substitutability

🎓: ★★★☆ 🏆 Can explain substitutability

Every instance of a subclass is an instance of the superclass, but not vice-versa. As a result, inheritance allows *substitutability*: the ability to substitute a child class object where a parent class object is expected.



An `AcademicStaff` is an instance of a `Staff`, but a `Staff` is not necessarily an instance of an `AcademicStaff`. i.e. wherever an object of the superclass is expected, it can be substituted by an object of any of its subclasses.

The following code is valid because an `AcademicStaff` object is substitutable as a `Staff` object.

```
Staff staff = new AcademicStaff(); // OK
```

But the following code is not valid because `staff` is declared as a `Staff` type and therefore its value may or may not be of type `AcademicStaff`, which is the type expected by variable `academicStaff`.

```
Staff staff;
...
AcademicStaff academicStaff = staff; // Not OK
```

Dynamic and Static Binding

: ★★★☆ Can explain dynamic and static binding

Dynamic binding (aka late binding): a mechanism where method calls in code are resolved at runtime, rather than at compile time.

Overridden methods are resolved using dynamic binding, and therefore resolves to the implementation in the actual type of the object.

Consider the code below. The declared type of `s` is `Staff` and it appears as if the `adjustSalary(int)` operation of the `Staff` class is invoked.

```
void adjustSalary(int byPercent) {
    for (Staff s: staff) {
        s.adjustSalary(byPercent);
    }
}
```

However, at runtime `s` can receive an object of any subclass of `Staff`. That means the `adjustSalary(int)` operation of the actual subclass object will be called. If the subclass does not override that operation, the operation defined in the superclass (in this case, `Staff` class) will be called.

Static binding (aka early binding): When a method call is resolved at compile time.

In contrast, overloaded methods are resolved using static binding.

Note how the constructor is overloaded in the class below. The method call `new Account()` is bound to the first constructor at compile time.

```
class Account {

    Account() {
        // Signature: ()
        ...
    }

    Account(String name, String number, double balance) {
        // Signature: (String, String, double)
        ...
    }
}
```

Similarly, the `calculateGrade` method is overloaded in the code below and a method call `calculateGrade("A1213232")` is bound to the second implementation, at compile time.

```
void calculateGrade(int[] averages) { ... }
void calculateGrade(String matric) { ... }
```

Polymorphism

What

: ★★★☆ Can explain OOP polymorphism

Polymorphism:

> The ability of different objects to respond, each in its own way, to identical messages is called polymorphism. -- Object-Oriented Programming with Objective-C, Apple

***Polymorphism* allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object.**

Assume classes `Cat` and `Dog` are both subclasses of the `Animal` class. You can write code targeting `Animal` objects and use that code on `Cat` and `Dog` objects, achieving possibly different results based on whether it is a `Cat` object or a `Dog` object. Some examples:

- Declare an array of type `Animal` and still be able to store `Dog` and `Cat` objects in it.
- Define a method that takes an `Animal` object as a parameter and yet be able to pass `Dog` and `Cat` objects to it.
- Call a method on a `Dog` or a `Cat` object as if it is an `Animal` object (i.e., without knowing whether it is a `Dog` object or a `Cat` object) and get a different response from it based on its actual class e.g., call the `Animal` class's method `speak()` on object `a` and get a `"Meow"` as the return value if `a` is a `Cat` object and `"Woof"` if it is a `Dog` object.

Polymorphism literally means "ability to take many forms".

How

🎓: ★★★☆ 🏆 Can explain how substitutability operation overriding, and dynamic binding relates to polymorphism

Three concepts combine to achieve polymorphism: *substitutability, operation overriding, and dynamic binding*.

- **Substitutability:** Because of substitutability, you can write code that expects objects of a parent class and yet use that code with objects of child classes. That is how polymorphism is able to *treat objects of different types as one type*.
- **Overriding:** To get polymorphic behavior from an operation, the operation in the superclass needs to be overridden in each of the subclasses. That is how overriding allows objects of different subclasses to *display different behaviors in response to the same method call*.
- **Dynamic binding**: Calls to overridden methods are bound to the implementation of the actual object's class dynamically during the runtime. That is how the polymorphic code can call the method of the parent class and yet execute the implementation of the child class.

Exercises

More

Miscellaneous

★★★☆ 🏆 Can answer frequently asked OOP questions

What is the difference between a Class, an Abstract Class, and an Interface?

- An interface is a behavior specification with no implementation.
- A class is a behavior specification + implementation.
- An abstract class is a behavior specification + a possibly incomplete implementation.

How does *overriding* differ from *overloading*?

Overloading is used to indicate that multiple operations do similar things but take different parameters. Overloaded methods have the same method name but different method signatures and possibly different return types.

Overriding is when a sub-class redefines an operation using the same method name and the same type signature. Overridden methods have the same name, same method signature, and same return type.

Review

★★★★ 🏆 Can combine some OOP concepts

...

Exercises