FREE eBook

LEARNING

oop

Free unaffiliated eBook created from
Stack Overflow contributors.

#oop

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: oop

It is an unofficial and free oop ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official oop.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with oop

Remarks

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Examples

Introduction

OOP - Object Oriented Programming is a vastly used programming paradigm in these days. In OOP, we model real world problems using Objects and there behaviors, in order to solve them, programmatically.

There are four main **OOP Concepts**

1. Inheritance
2. Polymorphism
3. Abstraction
4. Encapsulation

These four concepts together are used to develop programs in OOP.

There are various languages which supprots Object Oriented Programming. Most popular languages are

- C++
- Java
- C#
- Python (Python isn't fully Object Oriented, but has most of the OOP features of it)

OOP Introduction

Intoduction

Object Oriented Programming (mostly referred as OOP) is a programming paradigm for solving problems.
The beauty an OO (object oriented) program, is that we think about the program as a bunch of objects communicating with each other, instead of as a sequential script following specific orders.

There are lots of programming languages which support OOP, some of the popular are:

- Java

- C++
- c#

Python is also known to support OOP but it lacks a few properties.

OOP Terminology

The most basic term in OOP is a `class`.
A class is basically an *object*, which has a state and it works according to its state.

Another important term is an `instance`.
Think of a class as a template used to create instances of itself. The class is a template and the instance(s) is the concrete objects.

An instance created from the class *A* is usually referred to as from the 'type A', exactly like the type of *5* is *int* and the type of *"abcd"* is a *string*.

> An example of creating an instance named *insance1* of type (class) *ClassA*:
>
> ### Java
>
> ```
> ClassA instance1 = new ClassA();
> ```
>
> ### C++
>
> ```
> ClassA instance1;
> ```
>
> or
>
> ```
> ClassA *instance1 = new ClassA(); # On the heap
> ```
>
> ### Python
>
> ```
> instance1 = ClassA()
> ```

As you can see in the example above, in all cases the name of the class was mentioned and after it there was empty parentheses (except for the C++ where if they are empty the parentheses can be dropped). In these parentheses we can pass `arguments` to the `constructor` of our class.

A constructor is a method of a class which is called every time an instance is created. It can either take arguments or not. If the programmer does not specify any constructor for a class they build, an empty constructor will be created (a constructor which does nothing).
In most languages the constructor is defined as a method without defining its return type and with the same name of the class (example in a few sections).

An example of creating an instance named *b1* of type (class) *ClassB*. The constructor of *ClassB* takes one argument of type *int*:

Java

```
ClassA instance1 = new ClassA(5);
```

or

```
int i = 5;
ClassA instance1 = new ClassA(i);
```

C++

```
ClassA instance1(5);
```

Python

```
instance1 = ClassA(5)
```

As you can see, the process of creating an instance is very similar to the process of calling a function.

Functions vs Methods

Both functions and methods are very similar, but in Object Oriented Design (OOD) they each have their own meaning.
A method is an operation performed on an instance of a class. The method itself usually uses the state of the instance to operate.
Meanwhile, a function belongs to a class and not to a specific instance. This means it does not use the state of the class or any data stored in an instance.

From now on we will show our examples only in *Java* since OOP is very clear in this language, but the same principles work in any other OOP language.

In Java, a function has the word `static` in its definition, like so:

```
// File's name is ClassA
public static int add(int a, int b) {
    return a + b;
}
```

This means you can call it from anywhere in the script.

```
// From the same file
```

```
System.out.println(add(3, 5));

// From another file in the same package (or after imported)
System.out.println(ClassA.add(3, 5));
```

When we call the function from another file we use the name of the class (in Java this is also the name of the file) it belongs to, this gives the intuition that the function belongs to the class and not any of its instances.

In contrast, we can define a mehod in *ClassA* like so:

```
// File's name is ClassA
public int subtract(int a, int b){
    return a - b;
}
```

After this decleration we can call this method like so:

```
ClassA a = new ClassA();
System.out.println(a.subtract(3, 5));
```

Here we needed to create an instance of *ClassA* in order to call its method subtract. Notice we **CAN'T** do the following:

```
System.out.println(ClassA.subtract(3, 5));
```

This line will produce a compilation error complaining we called this *non-static* method without an instance.

Using the state of a class

Let's suppose we want to implement our *subtract* method again, but this time we always want to subtract the same number (for each instance). We can create the following class:

```
class ClassB {

    private int sub_amount;

    public ClassB(int sub_amount) {
        this.sub_amount = sub_amount;
    }

    public int subtract(int a) {
        return a - sub_amount;
    }

    public static void main(String[] args) {
        ClassB b = new ClassB(5);
        System.out.println(b.subtract(3)); // Ouput is -2
    }
```

```
}
```

When we run this code, a new instance named *b* of class *ClassB* is created and its constructor is fed with the value *5*.
The constructor now takes the given *sub_amount* and stores it as its own private field, also called *sub_amount* (this convention is very known in Java, to name the arguments the same as the fields).
After that, we print to the console the result of calling the method *subtract* on *b* with the value of *3*.

Notice that in the implementation of *subtract* we don't use `this.` like in the constructor.
In Java, `this` only needs to be written when there is another variable with the same name defined in that scope. The same works with Python's `self`.
So when we use *sub_amount* in subtract, we reference the private field which is different for each class.

Another example to emphasize.
Let's just change the main function in the above code to the following:

```
ClassB b1 = new ClassB(1);
ClassB b2 = new ClassB(2);

System.out.println(b1.subtract(10)); // Output is 9
System.out.println(b2.subtract(10)); // Output is 8
```

As we can see, *b1* and *b2* are independent and each have their own *state*.

Interfaces and Inheritance

An `interface` is a contract, it defines which methods a class will have and therefore its capabilities.
An interface does not have an implementation, it only defined what needs to be done.
An example in Java would be:

```
interface Printalbe {
    public void print();
}
```

The *Printalbe* interface defines a method called *print* but it doesn't give its implementation (pretty weird for Java). Every class which declares itself as `implementing` this interface must provide an implementation to the draw method. For example:

```
class Person implements Printalbe {

    private String name;

    public Person(String name) {
        this.name = name;
    }
```

```
    public void print() {
        System.out.println(name);
    }
}
```

If *Person* would declare itself as implementing *Drawable* but didn't provide an implementation to *print*, there would be a compilation error and the program wouldn't compile.

Inheritance is a term which points to a class *extending* another class. For example, let's say we now have a person who has an age. One way to implement a person like that would be to copy the *Person* class and write a new class called *AgedPerson* which has the same fields and methods but it has another property -age.
This would be awful since we duplicate our **entire** code just to add a simple feature to our class. We can use inheritance to inherit from *Person* and thus get all of its features, then enhance them with our new feature, like so:

```
class AgedPerson extends Person {

    private int age;

    public AgedPerson(String name, int age) {
        super(name);
        this.age = age;
    }

    public void print() {
        System.out.println("Name: " + name + ", age:" + age);
    }
}
```

There are a few new things going on:

- We used the saved word `extends` to indicate we are inheriting from Person (and also its implementation to *Printable*, so we do not need to declare `implementing Printable` again).
- We used the save word `super` to call *Person*'s constructor.
- We overridden the *print* method of *Person* with a new one.

This is getting pretty Java technical so I will not go any deeper into this topic. But I will mention that there a lot of extreme cases that should be learned about inheritance and interfaces before starting to use them. For example which methods and functions are inherited? What happens to private/public/protected fields when inheriting from a class? and so on.

Abstract Class

An `abstract class` is pretty advanced term in OOP which describes a combination of both interfaces and inheritance. It lets you write a class which has both implemented and unimplemented methods/functions in. In Java this is done by using the keyword `abstract` and I won't explain it more that a quick example:

```
abstract class AbstractIntStack {
```

```
    abstract public void push(int element);

    abstract public void pop();

    abstract public int top();

    final public void replaceTop(int element) {
        pop();
        push(element);
    }
}
```

Note: the `final` keyword states that you can't override this method when you inherit from this class. If a class is declared final, then no class can inherit from it at all.

Read Getting started with oop online: https://riptutorial.com/oop/topic/5081/getting-started-with-oop

Chapter 2: Abstraction

Examples

Abstraction - Introduction

Abstraction is one of the main concepts in **Object Oriented Programming (OOP)**. This is the process of hiding the implementation details for the outsiders while showing only essential details. In another words, Abstraction is a technique to arrange the complexity of a program.

There are two basic type of abstraction:

1. Control abstraction

   This is done using sub-routines and control flow. We can call another function/method/routine (sub-routine) from a function/method to do a specific task, where that sub-routine is abstract.

2. Data abstraction

   This is done through various data structures and their implementations. We can create our own data structures to store our data, while keeping the implementation abstract.

In *OOP* we use mix of control and function abstraction.

Access Modifiers

Access modifiers are used to control the access to an object or to a function/method. This is a main part of the concept of *Abstraction*.

Different programming languages use different access modifiers. Here are some examples:

- **Java**

  Java has 4 access modifiers.

  1. `private` - These attributes can be accessed only inside the class.
  2. `protected` - These attributes can be accessed by sub classes and classes from the same package.
  3. `package` - These attributes can be accessed by the classes within the same package only.
  4. `public` - These attributes can be accessed by everybody.

- **C++**

  C++ has 3 access modifiers.

  1. `private` - These attributes can be accessed only inside the class.

2. `protected` - These attributes can be accessed by derived classes.
3. `public` - These attributes can be accessed by everybody.

- **C#**

  C# has 5 access modifiers

  1. `private` - These attributes can be accessed only inside the class.
  2. `protected internal` - These attributes can be accessed by same assembly and derived classes.
  3. `protected` - These attributes can be accessed by derived classes.
  4. `public internal` - These attributes can be accessed by the classes within the same assembly.
  5. `public` - These attributes can be accessed by everybody.

Read Abstraction online: https://riptutorial.com/oop/topic/7324/abstraction

Chapter 3: Class

Examples

Introduction

Class is the piece of code where we define the attributes and/or behaviors of an object. You can define variables, constants, methods and constructors to the object, inside the class. In another words, class is the blueprint of an object.

Let's see a sample class in Java, which defines a (simple) Car:

```
public class Car {
  private Engine engine;
  private Body body;
  private Tire [] tire;
  private Interior interior;

  // Constructor
  public Car (Engine engine, Body body, Tire[] tires, Interior interior) {

  }

  // Another constructor
  public Car () {

  }

  public void drive(Direction d) {
    // Method to drive
  }

  public void start(Key key) {
    // Start
  }
}
```

This is just for an example. You can model real world object like this, as per your requirement.

Read Class online: https://riptutorial.com/oop/topic/7374/class

Chapter 4: Diamond problem

Examples

Diamond Problem - Example

Diamond problem is a common problem occurred in Object Oriented Programming, while using `multiple-inheritance`.

Consider the case where `class C`, is inherited from `class A` and `class B`. Suppose that both `class A` and `class B` have a method called `foo()`.

Then when we are calling the method `foo()`, compiler cannot identify the exact method we are trying to use

- `foo()` from `class A`
- `foo()` from `class B`

This is called the diamond problem basically. There are some variants of this problem. To avoid this, there are multiple approaches. **Java** doesn't allow multiple inheritance. Hence the problem is avoided. But C++ is allowing multiple inheritance, therefore you must be careful to use of multiple inheritance.

Read Diamond problem online: https://riptutorial.com/oop/topic/7319/diamond-problem

Chapter 5: Encapsulation

Examples

Information hiding

The state of an object at a given time is represented by the information that it holds at that point. In an OO language, the state is implemented as member variables.

In a properly designed object, the state can be changed only by means of calls to its methods and not by direct manipulation of its member variables. This is achieved by providing public methods that operate on the values of private member variables. The hiding of information in this manner is known as *encapsulation*.

Therefore, encapsulation ensures that private information is not exposed and cannot be modified except through calls to accessors and methods, respectively.

In the following example, you cannot set an `Animal` to be no longer hungry by changing the `hungry` private field; instead, you have to invoke the method `eat()`, which alters the state of the `Animal` by setting the `hungry` flag to `false`.

```
public class Animal {
    private boolean hungry;

    public boolean isHungry() {
        return this.hungry;
    }

    public void eat() {
        this.hungry = false;
    }
}
```

Read Encapsulation online: https://riptutorial.com/oop/topic/5958/encapsulation

Chapter 6: Inheritance

Remarks

Note: Multi-level inheritance is allowed in Java but not multiple inheritance. Find out more at http://beginnersbook.com/2013/04/oops-concepts/

Examples

Inheritance - Definition

Inheritance is one of the main concepts in **Object Oriented Programming (OOP)**. Using inheritance, we can model a problem properly and we can reduce the number of lines we have to write. Let's see inheritance using a popular example.

Consider you have to model animal kingdom (Simplified animal kingdom, of course. Biologists, pardon me) using OOP. There are a lots of species of animals, some have unique features, while some share same features.

There a are main families of animals. Let's say, `Mammals`, `Reptiles`.

Then we have children of those families. For an example,

- `Cat`, `Dog`, and `Lion` are mammals.
- `Cobra` and `Python` are reptiles.

Every animal shares some basic features like `eat`, `drink`, `move`. Hence we can say that we can have a parent called `Animal` from which they can inherit those basic features.

Then those families also shares some features. For an example reptiles use *crawling* to move. Every mammal is `fed milk` at early stages of life.

Then there are some unique features for each and every animal.

Consider if we are to create these animal species separately. We have to write same code again and again in every animal species. Instead of that, we use inheritance. We can model the Animal Kingdom as follows:

- We can have parent Object called `Animal`, which have basic features of all the animals.
- `Mammal` and `Reptile` (of course the other animal families also) objects with their common features while inheriting the basic features from parent object, `Animal`.
- Animal species objects: `Cat` and `Dog` inherits from `Mammal`object, `Cobra` and `Python` inherits from `Reptile` object, and so on.

In this form we can reduce the code we write, as we do not need to define basic features of Animals in each animal species, as we can define them in the `Animal` object and then inherit them. Same thing goes with the animal families.

Inheritance Example - Consider below two classes

Teacher Class:

```
class Teacher {
       private String name;
       private double salary;
       private String subject;
       public Teacher (String tname)  {
           name = tname;
       }
       public String getName()  {
           return name;
       }
       private double getSalary()  {
           return salary;
       }
       private String  getSubject()  {
            return  subject;
       }
    }
```

OfficeStaff Class:

```
class  OfficeStaff{
   private String name;
   private double salary;
   private String dept;
   public OfficeStaff (String sname)  {
      name = sname;
   }
   public String getName()  {
       return name;
   }
   private double  getSalary()  {
       return salary;
   }
   private String  getDept ()  {
       return dept;
   }
}
```

1. Both the classes share few common properties and methods. Thus repetition of code.
2. Creating a class which contains the common methods and properties.
3. The classes Teacher and OfficeStaff can inherit the all the common properties and methods from below Employee class.

Employee Class:

```
class Employee{
   private String name;
   private double salary;
   public Employee(String ename){
      name=ename;
   }
   public String getName(){
```

```
      return name;
   }
   private double getSalary(){
      return salary;
   }
}
```

4. Add individual methods and properties to it Once we have created a super class that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses
5. Any similar classes like Engineer, Principal can be generated as subclasses from the Employee class.
6. The parent class is termed super class and the inherited class is the sub class
7. A sub class is the specialized version of a super class – It inherits all of the instance variables and methods defined by the super class and adds its own, unique elements.
8. Although a sub class includes all of the members of its super class it can not access those members of the super class that have been declared as private.
9. A reference variable of a super class can be assigned to a reference to any sub class derived from that super class i.e. Employee emp = new Teacher();

Read Inheritance online: https://riptutorial.com/oop/topic/7321/inheritance

Chapter 7: Object

Examples

Introduction

Object is the base module in the **Object Oriented Programming (OOP)**. An Object can be a variable, data structure (like an array, map, etc), or even a function or method. In OOP, we model real world objects like animals, vehicles, etc.

An object can be defined in a class, which can be defined as the blueprint of the object. Then we can create instances of that class, which we call objects. We can use these objects, their methods and variables in our code after.

Read Object online: https://riptutorial.com/oop/topic/7377/object

Chapter 8: Polymorphism

Examples

Introduction

Polymorphism is one of the basic concepts in **OOP (Object Oriented Programming)**. Main idea of the polymorphism is that an object have the ability to take on different forms. To achieve that (polymorphism), we have two main approaches.

1. Method overloading
    - Occures when there are two or more methods with the same name, with different input parameters. **The return type should be the same for all the methods with the same name**
2. Method overriding
    - Occures when child object uses same method definition (same name with same parameters), but have different implementations.

Using these two approaches we can use the same method/function to behave differently. Let's see more details on this in following examples.

Method Overloading

Method overloading is the way of using polymorphism inside a class. We can have two or more methods inside the same class, with different input parameters.

Difference of input parameters can be either:

- Number of parameters
- Type of parameters (Data type)
- Order of the parameters

Let's take a look at them separately (These examples in java, as I am more familiar with it - Sorry about that):

1. Number of Parameters

```
public class Mathematics {
    public int add (int a, int b) {
        return (a + b);
    }

    public int add (int a, int b, int c) {
        return (a + b + c);
    }
```

```
     public int add (int a, int b, int c, int c) {
         return (a + b + c + d);
     }
 }
```

Look carefully, you can see the method's return type is the same - `int`, but theree of these methods having different number of inputs. This is called as method over loading with different number of parameters.

PS: This is a just an *example*, there's no need to define add functions like this.

2. Type of parameters

```
public class Mathematics {
    public void display (int a) {
        System.out.println("" + a);
    }

    public void display (double a) {
        System.out.println("" + a);
    }

    public void display (float a) {
        System.out.println("" + a);
    }
}
```

Note that every method has the same name and same return type, while they have different input data types.

PS: This example is only for explaining purpose only.

3. Order of the parameters

```
public class Mathematics {
    public void display (int a, double b) {
        System.out.println("Numbers are " + a + " and " + b);
    }

    public void display (double a, int b) {
        System.out.println("Numbers are " + a + " and " + b);
    }
}
```

PS: This example is also for explaining purpose only.

Method Overriding

Method overriding is the way of using polymorphism between classes. if one class is inherited from another, the former (sub class) can override the latter's (super class's) methods, and change the implementation.

this is used where the super class defines the more general implementation of the method while the sub class uses a more specific one.

Consider following example:

We have a class for Mammals:

```
class Mammal {
  void whoIam () {
    System.out.println("I am a Mammal");
  }
}
```

Then we have a class for Dog, which is a Mammal:

```
class Dog extends Mammal {
  @Override
  void whoIam () {
    super.whoIam();
    System.out.println("I am a Dog!");
  }
}
```

In this example, we define `whoIam()` method in `Mammal` class, where the mammal say it is a Mammal. But this is a general term, as there are a lot of Mammals out there. Then we can inherit `Dog` class from `Mammal` class, as Dog is a Mammal. But, to be more specific, Dog is a Dog as well as a Mammal. Hence, Dog should say, `I am a Mammal` and also `I am a Dog`. Hence we can **Override** the `whoIam()` method in super class (`Mammal` class, that is) from sub class (the `Dog` class).

We can also call the super class's method using `super.whoIam()` in Java. Then, the `Dog` will behave like a Dog, while also behaving like a Mammal.

Read Polymorphism online: https://riptutorial.com/oop/topic/7323/polymorphism

Credits

S. No	Chapters	Contributors
1	Getting started with oop	Alon Alexander, Community, James, Thisaru Guruge
2	Abstraction	davioooh, Thisaru Guruge
3	Class	Thisaru Guruge
4	Diamond problem	Thisaru Guruge
5	Encapsulation	davioooh, jeyoung
6	Inheritance	SRENG Khorn, Thisaru Guruge
7	Object	Thisaru Guruge
8	Polymorphism	Fantasy Pollock, murthy, Thisaru Guruge