

Principles of Java Class Design

Class is a very basic unit of object-oriented programming. It is the primary build structure from which the instances of it, called the *object*, are extracted. The principles of class design, however preliminary it may seem, are decisive of the foundation of an application. Experienced programmers follow principles to design classes that live beyond their immediate need. Building an robust application requires adherence to many principles, such as general design principles, programming idioms, thumb rule, tricks of the trade, cautionary measure, and so forth. Class design is just one among them. A good style of programming is essential for practical reasons, but it is even more important in object-oriented programming because much of the benefit of the object-oriented approach is predicated on producing reusable, extensible, and understandable programs. The article explains some of the key areas of class design with respect to object-oriented languages in general, and [Java](#) class design.

Aspects of Java Class Design

Class design that follows established principles is usually correct, reusable, extensible, and easy to maintain. But, that does not mean they are providence. A situation may arise where normal principles find it inadequate to fit in. In such a case, the purpose or objective should be the basic ingredient of design decision. But, such a situation may arise once in a blue moon. It is always a good idea to begin with a proper design guideline rather than grope in the dark and become sorry in the later phases of development.

Before designing a class, we must understand that a class is an abstraction of a certain aspect of a problem. The objective is to filter those aspects that are important for the purpose and ignore those that are unimportant. Sometimes, given a problem, one may find it difficult to identify aspects that are the right candidate to be classified. After all, abstraction in computing is a poor representation of reality; even establishing a simple relationship can be quite complex. Still, they do represent because we insist. Depending upon the complexity of the problem, it can even be a challenge for a seasoned programmer. Perhaps, a way out of the situation is to focus on the purpose, because it is the purpose that determines what is to be taken into account and what to suppress. Also, depending upon the purpose to be served, a variety of abstractions of the same thing is possible. For example, a *Media* class construct may be designed differently depending upon the type of media it targets, such as print media or screen.

Another issue is that abstractions are always incomplete and inaccurate. It is our responsibility to create a design that pushes it into right track. This also means that trying to create an absolutely correct design is an Utopian dream. It's not going to happen. What, at best, we can do is create what is adequate, right for the purpose, using the tools at hand. An abstraction can never express the cobweb of reality. Something like the spoken languages, however sophisticated, rich it may seem, can relay only a partial aspect of our thought.

Class in a Nutshell

A class can be described as a collection of objects (attributes), behavior (methods), or relationship with other objects and have common semantics. For example, *Employee*, *Window*, *Vehicle*, *Shape*, *Account*, *Book*, *Transaction*, *Process* and so forth are appropriate candidates to be designed as classes. Sometimes, classes derive their identity in relation to another class, technically called *inheritance*. Their relationship is named as parent-child or super-subclass relationship.

Naming a Class

Naming a class is the first step to designing a class. It must reflect the purpose and meaning of its existence. The name itself gives a clue to its capability. Unlike other object programming languages that embraced a laconic naming scheme, Java is verbose. Some of the class names in the Java API library are quite a mouthful. Here are a few of them, just as an example:

AbstractQueuedLongSynchronizer, *AdapterNonExistentHelper*, *AlgorithmParameterGeneratorSpi*, *AnnotationTypeMismatchException*, *AtomicMoveNotSupportedException*.

Although Unix users and C/C++ programmers would disagree, there is a point in adhering to such verbosity. The name itself becomes the documentation by providing an excellent introduction to its meaning and utility. However, names do not only determine the individuality of a class, because they may be identical. For example, the class *java.util.List* (it is an interface, same as a class from the perspective of abstraction) is quite different from *java.awt.List* even though they have

the same name. There are, no doubt, certain similarities; otherwise, the name would not have been the same. But, their purposes are utterly different. One represents an ordered collection of elements, another a GUI component that holds a scrolling list of text items. So, the parameters of distinction among classes includes the name along with its attributes and behavior pattern.

Encapsulation

Encapsulation, or *Information hiding*, is the basic paradigm of a good class design. It is a way to hide internal information of the class from external view by treating a class as a black box. For example, a class and its attributes declared as:

```
public class Person{
    public String name;
    public String phone;
}
```

make it accessible as,

```
Person p=new Person();
p.name="Zairo";
p.phone=1234567890;
```

Java allows accessing attributes with object names if they are declared as public. This is a bad idea under normal circumstances and violates the principle of information hiding. Instead, the visibility of the attributes should be restricted by declaring them as **private** or **protected** and **creating a set of accessors** and mutator methods, called **getters and setters**. There are several advantages of using getters and setters.

```
public class Person{
    private String name;
    private String phone;
    public String getName() { return name; }
    public void setName(String name)
        { this.name=name; }
    public String getPhone() { return phone; }
    public void setPhone(String phone)
        { this.phone=phone; }
}
```

Because it leverages the field access control, it is easy to make the class thread-safe if we want to later. Also, we can better handle any changes made to the property's implementation. These changes will not be visible to the object's user,

```
public class Person{
    // ...
    private Integer phone;
    // ...

    public String getPhone()
        { return phone.toString() }
    public void setPhone(String phone)
        { this.phone=Integer.parseInt(phone); }
}
```

Coupling

Loose coupling is another aspect to be taken very seriously when designing a class and its relationships with other classes in the family. It defines the degree to which a class depends upon others. Two classes are considered independent if they can act separately. In practice, classes rarely behave in an independent manner. They must interact to produce the desired external behavior. The more interconnection between classes, the more dependent they are, because more knowledge about the class is required to understand and deal with other classes. This is called *tightly coupled*. And, tightly coupled designs cannot leverage re-usability and extensibility. Therefore, the class design should be made in a way that it is as loosely coupled as possible. There are couple of ways to achieve this in Java, so that interactivity is not marred by loose coupling. What we can do is create a pure abstraction class that handles the interaction between two classes. Another approach is to delegate responsibility of interaction to an isolated class that we do not intend to make reusable.

Two classes are **tightly coupled** if,

- The function calls between two classes involve **passing large chunks of shared data**.
- Interaction occurs through the use of **some shared data**.

Cohesion

To put it in a simple manner, if the functions of a class co-operate with each other to attain a single objective, the class design is said to be cohesive; otherwise, if they are too varied, like a conflicting interest among family members, the class design is invariably poor. Therefore, it is a property that specifies how tightly bound the elements of a class are. Strong cohesion is a desirable phenomenon where all the elements are together to support a well-defined abstraction. Cohesion gets spoiled if too much functionality is added. **A class with limited functionality** is not only **manageable** but also **conducive to strong cohesion**.

Some Other Java Class Design Guidelines

There are many more guidelines, but these few are often found to be quite helpful to begin with.

- **Only the operations** that are needed **by the user of the class** should be made **public**.
- An instance of a class **should not** send messages **directly** to components of another class.
- Operations defined in the class should be such that they operate on the data defined on the class.
- The public interface of a class should contain only operations defined on the class.
- Classes should be **the least dependent** as possible.
- A class hierarchy in the inheritance should be drawn based upon their natural relationship and never imposed. The topmost class should be an abstract class or an interface.
- The number of arguments passed to a method and its size should be small.

Conclusion

The class definition makes up the most of object-oriented design. When we talk of principles in this respect, it associates with the language design principles where Java is one of the many tools of implementation. Class design has a major impact on the overall quality of the software design. It must be remembered that good quality classes are always reusable or extensible. IMHO, a good design always begins with a lot of common sense and practice; principles and guidelines are important ingredients of it. What Frankenstein one is going to make is purely at the discretion of the developer.

What is OOPS in JAVA?

Object Oriented Programming popularly known as OOP, is used in a modern programming language like Java

Core OOPS concepts are

1) Class

The class is a group of similar entities. It is only an logical component and not the physical entity. ex, if you had a class called “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc. Its **properties**(data) can be **price** or **speed** of these cars. While the **methods** may be performed with these cars are **driving, reverse, braking** etc.

2) Object

An object can be defined as an instance of a class, and there can be multiple instances of a class in a program. An Object contains both the data and the function, which operates on the data. For example - chair, bike, marker, pen, table, car, etc.

3) Inheritance

Inheritance is an OOPS concept in which one object acquires the properties and behaviors of the parent object. It's creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.

4) Polymorphism

Polymorphism refers to the ability of a **variable**, **object** or **function** to take on multiple forms. ex, in English, the verb "run" has a different meaning if you use it with "a laptop", "a foot race", and "business". Here, we understand the meaning of "run" based on the other words used along with it. The same also applied to Polymorphism.

5) Abstraction

An abstraction is an act of representing essential features without including background details. It is a technique of creating a new data type that is suited for a specific application. ex, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

6) Encapsulation

Encapsulation is an OOP technique of wrapping the data and code. In this OOPS concept, the variables of a class are always hidden from other classes. It can only be accessed using the methods of their current class. For example - in school, a student cannot exist without a class.

7) Association

Association is a relationship between two objects. It defines the diversity between objects. In association concept, all object have their separate lifecycle, and there is no owner. Ex, many students can associate with one teacher while one student can also associate with multiple teachers.

8) Aggregation

All objects have their separate lifecycle. However, there is ownership such that child object can't belong to another parent object. Ex, consider class/objects department and teacher. Here, a single teacher can't belong to multiple departments, but even if we delete the department, the teacher object will never be destroyed.

9) Composition

A composition is a specialized form of Aggregation. It is also called "death" relationship. Child objects do not have their lifecycle so when parent object deletes all child object will also delete automatically. For that, let's take an example of House and rooms. Any house can have several rooms. One room can't become part of two different houses. So, if you delete the house room will also be deleted.

Advantages of OOPS:

- OOP offers easy to understand and a clear modular structure for programs.

- Objects created for Object-Oriented Programs can be reused in other programs. Thus it saves significant development cost.
- Large programs are difficult to write, but if the development and designing team follow OOPS concept then they can better design with minimum flaws.
- It also enhances program modularity because every object exists independently.

Comparison of OOPS with other programming styles with help of an Example

Let's understand with example how OOPs is different than other programming approaches. Programming languages can be classified into 3 primary types

1. **Unstructured Programming Languages:** The most primitive of all programming languages having sequentially flow of control. Code is repeated through out the program
2. **Structured Programming Languages:** Has non-sequentially flow of control. Use of functions allows for re-use of code.
3. **Object Oriented Programming:** Combines Data & Action Together.

Let's understand these 3 types with an example.

Suppose you want to create a Banking Software with functions like

1. Deposit
2. Withdraw
3. Show Balance

Unstructured Programming Languages

The earliest of all programming language were unstructured programming language.

A very elementary code of banking application in unstructured Programming language will have two variables of one account number and another for account balance.

```
int account_number=20;  
int account_balance=100;
```

Suppose deposit of 100 dollars is made.

```
account_balance=account_balance+100
```

Next you need to display account balance.

```
printf("Account Number=%d,account_number)  
printf("Account Balance=%d,account_balance)
```

Now the amount of 50 dollars is withdrawn.

```
account_balance=account_balance-50
```

Again, you need to display the account balance.

```
printf("Account Number=%d,account_number)  
printf("Account Balance=%d,account_balance)
```





Diagram illustrating Unstructured Programming. The code shows a sequence of operations on account data, with the same printing and calculation lines repeated multiple times. A red box highlights the text "unstructured Programming same code is repeated", with green dotted arrows pointing to the repeated code blocks.

```

int account_number = 20;
int account_balance = 100;

account_balance = account_balance + 100
printf("Account Number = %d", account_number)
printf("Account Balance = %d", account_balance)

account_balance = account_balance - 50
printf("Account Number = %d", account_number)
printf("Account Balance = %d", account_balance)

account_balance = account_balance - 10
printf("Account Number = %d", account_number)
printf("Account Balance = %d", account_balance)

```

For any further deposit or withdrawal operation – you will code repeat the same lines again and again.

Structured Programming

With the arrival of Structured programming repeated lines on the code were put into structures such as functions or methods. Whenever needed, a simple call to the function is made.

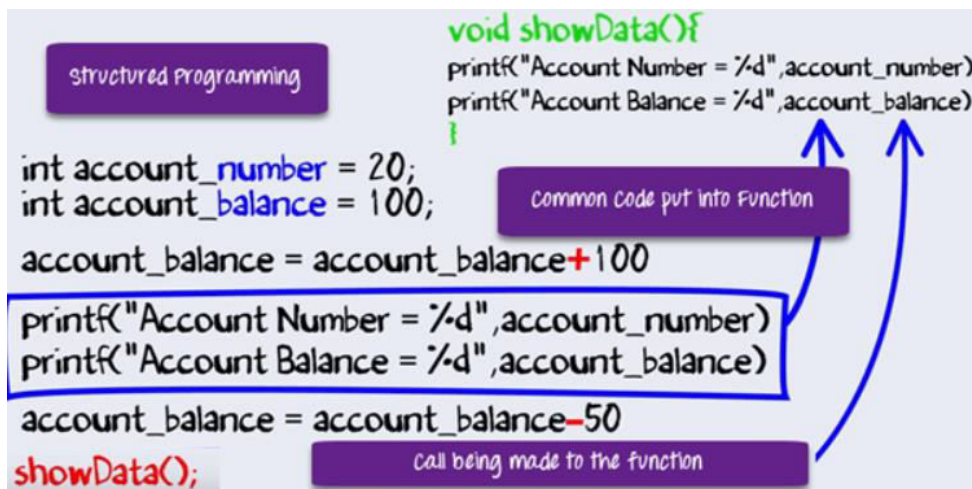


Diagram illustrating Structured Programming. The code is organized into a function `void showData()` which contains the repeated printing and calculation logic. A call to `showData();` is made in the main code. Annotations include: "structured Programming" (purple box), "common code put into Function" (purple box pointing to the function definition), and "call being made to the function" (purple box pointing to the function call).

```

void showData(){
    printf("Account Number = %d", account_number)
    printf("Account Balance = %d", account_balance)
}

int account_number = 20;
int account_balance = 100;

account_balance = account_balance + 100

printf("Account Number = %d", account_number)
printf("Account Balance = %d", account_balance)

account_balance = account_balance - 50

showData();

```

Object-Oriented Programming

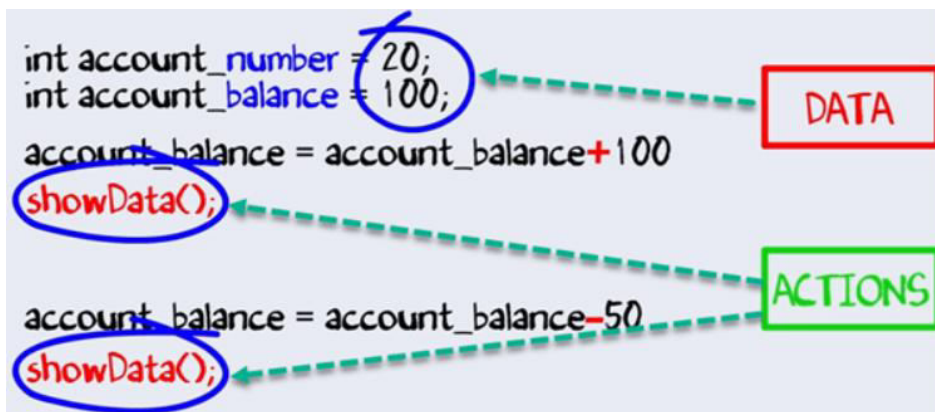
In our program, we are dealing with data or performing specific operations on the data.

In fact, having data and performing certain operation on that data is very basic characteristic in any software program.

Experts in Software Programming thought of combining the Data and Operations. Therefore, the birth of Object Oriented Programming which is commonly called OOPS.

The same code in OOPS will have same data and some action performed on that data.

```
Class Account{  
    int account_number;  
    int account_balance;  
public void showdata(){  
    system.out.println("Account Number"+account_number)  
    system.out.println("Account Balance"+ account_balance)  
}  
}
```



By combining data and action, we will get many advantages over structural programming viz,

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

They are discussed in greater details in succeeding tutorials