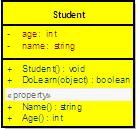
Object Oriented Programming Designs and Principle

Introduction  
Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. Programming techniques may include features such as abstraction, encapsulation, inheritance, and polymorphism.

## Building Blocks

Object  
  
basically an object is anything that is identifiable as a single material item. You can see around and find many objects like Camera, Monitor, and Laptop etc. In OOP perspective, an object is nothing but an instance of a class that contains real values instead of variables  
  
Class  
  
A *class* is simply a representation of a type of *object*. It is the blueprint, or plan, or template that describes the details of an *object*. A class is the blueprint from which the individual objects are created. *Class* is composed of three things: a name, attributes, and operations.



public class Student{}

Student objectStudent = new Student();

### Abstract class

Abstract classes, which declared with the abstract keyword, cannot be instantiated. It can only be used as a super-class for other classes that extend the abstract class. Abstract class is the concept and implementation gets completed when it is being realized by a subclass. In addition to this a class can inherit only from one abstract class (but a class may implement many interfaces) and must override all its methods/properties that are declared to be abstract and may override virtual methods/ properties. Abstract classes are ideal when implementing frameworks. As an example, LoggerBase below.

Hide   Shrink Image 8 for Introduction to Object Oriented Programming Concepts (OOP) and More   Copy Code

public abstract class LoggerBase

{

/// *<summary>*

/// *field is private, so it intend to use inside the class only*

/// *</summary>*

private log4net.ILog logger = null;

/// *<summary>*

/// *protected, so it only visible for inherited class*

/// *</summary>*

protected LoggerBase()

{

*// The private object is created inside the constructor*

logger = log4net.LogManager.GetLogger(this.LogPrefix);

*// The additional initialization is done immediately after*

log4net.Config.DOMConfigurator.Configure();

}

/// *<summary>*

/// *When you define the property as abstract,*

/// *it forces the inherited class to override the LogPrefix*

/// *So, with the help of this technique the log can be made,*

/// *inside the abstract class itself, irrespective of it origin.*

/// *If you study carefully you will find a reason for not to have "set" method here.*

/// *</summary>*

protected abstract System.Type LogPrefix

{

get;

}

/// *<summary>*

/// *Simple log method,*

/// *which is only visible for inherited classes*

/// *</summary>*

/// *<param name="message"></param>*

protected void LogError(string message)

{

if (this.logger.IsErrorEnabled)

{

this.logger.Error(message);

}

}

/// *<summary>*

/// *Public properties which exposes to inherited class*

/// *and all other classes that have access to inherited class*

/// *</summary>*

public bool IsThisLogError

{

get

{

return this.logger.IsErrorEnabled;

}

}

}

The idea of having this class as an abstract is to define a framework for exception logging. This class will allow all subclass to gain access to a common exception logging module and will facilitate to easily replace the logging library. By the time you define the LoggerBase, you wouldn’t have an idea about other modules of the system. But you do have a concept in mind and that is, if a class is going to log an exception, they have to inherit the LoggerBase. In other word the LoggerBase provide a framework for exception logging.

Apart from these you can also have virtual methods defined in an abstract class. The virtual method may have its default implementation, where a subclass can override it when required.

Abstract classes let you define some behaviors; they force your subclasses to provide others. For example, if you have an application framework, an abstract class can be used to provide the default implementation of the services and all mandatory modules such as event logging and message handling etc. This approach allows the developers to develop the application within the guided help provided by the framework.

However, in practice when you come across with some application-specific functionality that only your application can perform, such as startup and shutdown tasks etc. The abstract base class can declare virtual shutdown and startup methods. The base class knows that it needs those methods, but an abstract class lets your class admit that it doesn't know how to perform those actions; it only knows that it must initiate the actions. When it is time to start up, the abstract class can call the startup method. When the base class calls this method, it can execute the method defined by the child class.

Interface

In summary the Interface separates the implementation and defines the structure, and this concept is very useful in cases where you need the implementation to be interchangeable. Apart from that an interface is very useful when the implementation changes frequently. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can contain properties (which are also implicitly public and abstract). Interface definition begins with the keyword interface. An interface like that of an abstract class cannot be instantiated.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class. In addition to this an interfaces can inherit other interfaces.

Hide   Copy Code

public interface ILogger

{

bool IsThisLogError { get; }

}

difference between a Class and an Interface

In .NET/ C#, a *class* can be defined to implement an *interface* and also it supports multiple implementations. When a *class* implements an *interface*, an *object* of such *class* can be encapsulated inside an *interface*.

If MyLogger is a class, which implements ILogger*,* there we can write

Hide   Copy Code

ILogger log = new MyLogger();

A *class* and an *interface* are two different types (conceptually). Theoretically a *class* emphasis the idea of encapsulation, while an *interface* emphasis the idea of abstraction (by suppressing the details of the implementation). The two poses a clear separation from one to another. Therefore it is very difficult or rather impossible to have an effective meaningful comparison between the two, but it is very useful and also meaningful to have a comparison between an interface and an abstract class.

Implicit and Explicit Interface Implementations

As mentioned before .Net support multiple implementations, the concept of implicit and explicit implementation provide safe way to implement methods of multiple interfaces by hiding, exposing or preserving identities of each of interface methods, even when the method signatures are the same.

Let's consider the interfaces defined below.

Hide   Copy Code

interface IDisposable

{

void Dispose();

}

Here you can see that the classStudent has implicitly and explicitly implemented the method named Dispose() via Dispose and IDisposable.Dispose.

Hide   Copy Code

class Student : IDisposable

{

public void Dispose()

{

Console.WriteLine("Student.Dispose");

}

void IDisposable.Dispose()

{

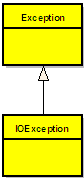
Console.WriteLine("IDisposable.Dispose");

}

}

### What is Inheritance?

The ability of a new class to be created, from an existing class by extending it, is called inheritance.



Hide   Copy Code

public class Exception

{

}

public class IOException : Exception

{

}

According to the above example the new class (IOException), which is called the derived class or subclass, inherits the members of an existing class (Exception), which is called the base class or super-class. The class IOException can extend the functionality of the class Exception by adding new types and methods and by overriding existing ones.

Inheritance is a mechanism of acquiring the features and behaviors of a class by another class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. Inheritance implements the IS-A relationship.

For example, mammal IS-A animal, dog IS-A mammal; Hence dog IS-A animal as well.

Different Types of Inheritance

OOPs support the six different types of inheritance as given below :

1. Single inheritance

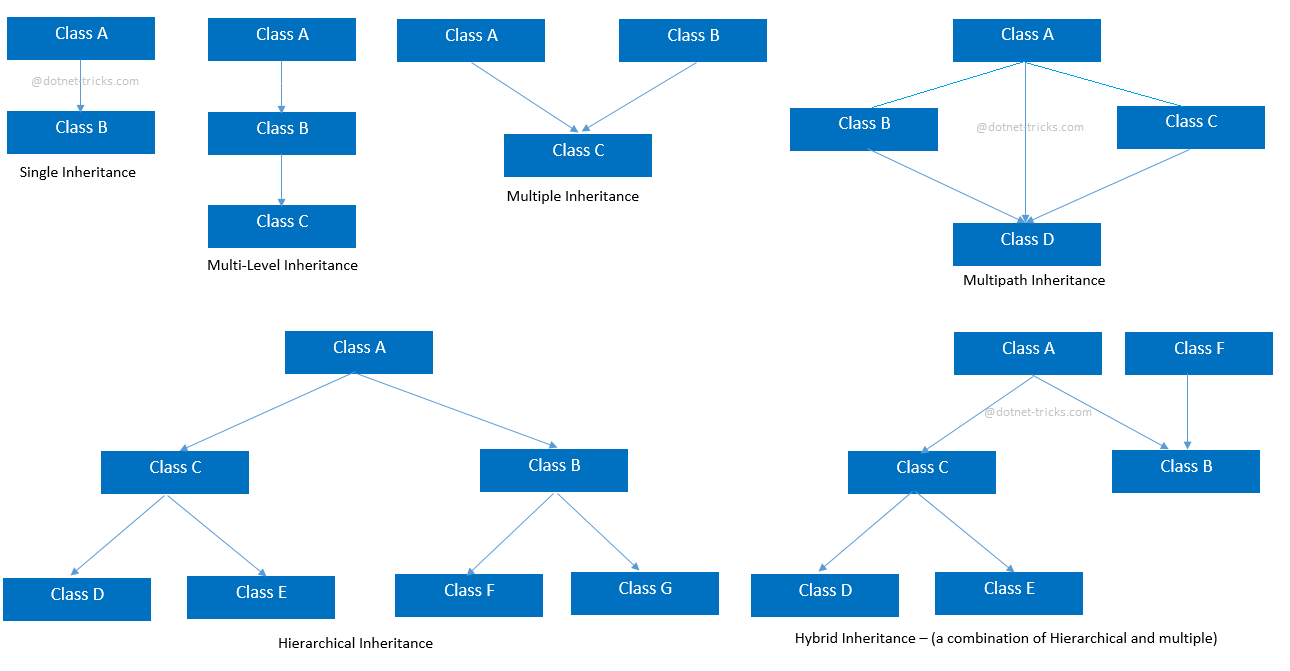
2. Multi-level inheritance

3. Multiple inheritance

4. Multipath inheritance

5. Hierarchical Inheritance

6. Hybrid Inheritance



1. Single inheritance

In this inheritance, a derived class is created from a single base class.

In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behaviour of the parent class A.

2. Multi-level inheritance

In this inheritance, a derived class is created from another derived class.

In the given example, class c inherits the properties and behavior of class B and class B inherits the properties and behavior of class B. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.

3. Multiple inheritance

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F# etc. and Java Language.

In the given example, class c inherits the properties and behavior of class B and class A at same level. So, here A and Class B both are the parent classes for Class C.

4. Multipath inheritance

In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes. This inheritance is not supported by .NET Languages like C#, F# etc.

In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherits the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it Multipath inheritance.

5. Hierarchical Inheritance

In this inheritance, more than one derived classes are created from a single base class and futher child classes act as parent classes for more than one child classes.

In the given example, class A has two childs class B and class D. Further, class B and class C both are having two childs - class D and E; class F and G respectively.

6. Hybrid inheritance

This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.

Since .NET Languages like C#, F# etc. does not support multiple and multipath inheritance. Hence hybrid inheritance with a combination of multiple or multipath inheritances is not supported by .NET Languages.

Behavioural and Implementation Inheritance

Interfaces are a mechanism for behavioural inheritance, whereas abstract classes are used for implementation inheritance. The bottom line is that interface language constructs provide behavioral interfaces, but no implementation, whereas abstract classes may provide both interfaces and implementation.

### Abstraction In object-oriented software, complexity is managed by using abstraction. Abstraction is a process that involves identifying the critical behaviour of an object and eliminating irrelevant and complex details. A well thought-out abstraction is usually simple, and easy to use in the perspective of the user, the person who is using your object.

### Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. Abstraction is essential in the construction of programs. It places the emphasis on what an object is or does rather than how it is represented or how it works. Thus, it is the primary means of managing complexity in large programs.

### // C# program to calculate the area

### // of a square using the concept of

### // data abstraction

### using System;

### namespace Demoabstraction {

### // abstract class

### abstract class Shape {

### // abstract method

### public abstract int area();

### }

### // square class inherting

### // the Shape class

### class Square : Shape {

### // private data member

### private int side;

### // method of square class

### public Square(int x = 0)

### {

### side = x;

### }

### 

### // overriding of the abstract method of Shape

### // class using the override keyword

### public override int area()

### {

### Console.Write("Area of Square: ");

### return (side \* side);

### }

### }

### // Driver Class

### class GFG {

### // Main Method

### static void Main(string[] args)

### {

### // creating reference of Shape class

### // which refer to Square class instance

### Shape sh = new Square(4);

### // calling the method

### double result = sh.area();

### Console.Write("{0}", result);

### }

### }

### }

##### Advantages of Abstraction

* It reduces the complexity of viewing the things.
* Avoids code duplication and increases reusability.
* Helps to increase security of an application or program as only important details are provided to the user.

### Abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces

### Encapsulation(or Information Hiding)

Encapsulation is a method for protecting data from unwanted access or alteration by packaging it in an object where it is only accessible through the object's interface. Encapsulation are often referred to as information hiding. But both are different. Infact information hiding is actually the result of Encapsulation. Encapsulation makes it possible to separate an object's implementation from its original behaviour - to restrict access of its internal data. This restriction facilitate certain details of an object;s behavior to be hidden. This allows to protect an object's integral state from corruption by its user.

The encapsulation is the inclusion-within a program object-of all the resources needed for the object to function, basically, the methods and the data. In OOP the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties. A class is kind of a container or capsule or a cell, which encapsulate a set of methods, attribute and properties to provide its indented functionalities to other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system. That idea of encapsulation is to hide how a class does its business, while allowing other classes to make requests of it.

Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member. C# supports the following access specifiers −

* Public
* Private
* Protected
* Internal
* Protected internal

## Public Access Specifier

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

The following example illustrates this −

[Live Demo](http://tpcg.io/K4GlIi)

using System;

namespace RectangleApplication {

class Rectangle {

//member variables

public double length;

public double width;

public double GetArea() {

return length \* width;

}

public void Display() {

Console.WriteLine("Length: {0}", length);

Console.WriteLine("Width: {0}", width);

Console.WriteLine("Area: {0}", GetArea());

}

}//end class Rectangle

class ExecuteRectangle {

static void Main(string[] args) {

Rectangle r = new Rectangle();

r.length = 4.5;

r.width = 3.5;

r.Display();

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Length: 4.5

Width: 3.5

Area: 15.75

In the preceding example, the member variables length and width are declared public, so they can be accessed from the function Main() using an instance of the Rectangle class, named r.

The member function Display() and GetArea() can also access these variables directly without using any instance of the class.

The member functions Display() is also declared public, so it can also be accessed from Main() using an instance of the Rectangle class, named r.

## Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

The following example illustrates this −

[Live Demo](http://tpcg.io/rIUpfp)

using System;

namespace RectangleApplication {

class Rectangle {

//member variables

private double length;

private double width;

public void Acceptdetails() {

Console.WriteLine("Enter Length: ");

length = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Enter Width: ");

width = Convert.ToDouble(Console.ReadLine());

}

public double GetArea() {

return length \* width;

}

public void Display() {

Console.WriteLine("Length: {0}", length);

Console.WriteLine("Width: {0}", width);

Console.WriteLine("Area: {0}", GetArea());

}

}//end class Rectangle

class ExecuteRectangle {

static void Main(string[] args) {

Rectangle r = new Rectangle();

r.Acceptdetails();

r.Display();

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Enter Length:

4.4

Enter Width:

3.3

Length: 4.4

Width: 3.3

Area: 14.52

In the preceding example, the member variables length and width are declared private, so they cannot be accessed from the function Main(). The member functions AcceptDetails() and Display() can access these variables. Since the member functions AcceptDetails() and Display() are declared public, they can be accessed from Main() using an instance of the Rectangle class, named r.

## Protected Access Specifier

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

## Internal Access Specifier

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

The following program illustrates this −

[Live Demo](http://tpcg.io/m3ixfT)

using System;

namespace RectangleApplication {

class Rectangle {

//member variables

internal double length;

internal double width;

double GetArea() {

return length \* width;

}

public void Display() {

Console.WriteLine("Length: {0}", length);

Console.WriteLine("Width: {0}", width);

Console.WriteLine("Area: {0}", GetArea());

}

}//end class Rectangle

class ExecuteRectangle {

static void Main(string[] args) {

Rectangle r = new Rectangle();

r.length = 4.5;

r.width = 3.5;

r.Display();

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Length: 4.5

Width: 3.5

Area: 15.75

In the preceding example, notice that the member function GetArea() is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is private.

## Protected Internal Access Specifier

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

##### Encapsulation vs Data Abstraction

* [Encapsulation](https://www.geeksforgeeks.org/c-encapsulation/)is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
* While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing to the user and hiding the details of implementation.

interface IDisposable

{

void Dispose();

}

Hide   Copy Code

class Student : IDisposable

{

void IDisposable.Dispose()

{

Console.WriteLine("IDisposable.Dispose");

}

}

## **Polymorphism**

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

## **Static Polymorphism or Early Binding or Compile Time Polymorphism**

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

* Method overloading
* Operator overloading

We discuss operator overloading in next chapter.

### What is Method Overloading?

Method overloading is the ability to define several methods all with the same name.

Hide   Copy Code

public class MyLogger

{

public void LogError(Exception e)

{

*// Implementation goes here*

}

public bool LogError(Exception e, string message)

{

*// Implementation goes here*

}

}

### 4.19. What is Operator Overloading?

The operator overloading (less commonly known as ad-hoc polymorphisms) is a specific case of polymorphisms in which some or all of operators like +, - or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.

Hide   Copy Code

public class Complex

{

private int real;

public int Real

{ get { return real; } }

private int imaginary;

public int Imaginary

{ get { return imaginary; } }

public Complex(int real, int imaginary)

{

this.real = real;

this.imaginary = imaginary;

}

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

}

}

I above example I have overloaded the plus operator for adding two complex numbers. There the two properties named Real and Imaginary has been declared exposing only the required “get” method, while the object’s constructor is demanding for mandatory real and imaginary values with the user defined constructor of the class.

## **Dynamic Polymorphism or Late Binding or Dynamic Binding**

## Run time polymorphism - It is achieved by overriding virtual functions

### 4.20. What is Method Overriding?

Method overriding is a language feature that allows a subclass to override a specific implementation of a method that is already provided by one of its super-classes.

A subclass can give its own definition of methods but need to have the same signature as the method in its super-class. This means that when overriding a method the subclass's method has to have the same name and parameter list as the super-class' overridden method.

Hide   Shrink Image 10 for Introduction to Object Oriented Programming Concepts (OOP) and More   Copy Code

using System;

public class Complex

{

private int real;

public int Real

{ get { return real; } }

private int imaginary;

public int Imaginary

{ get { return imaginary; } }

public Complex(int real, int imaginary)

{

this.real = real;

this.imaginary = imaginary;

}

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

}

public override string ToString()

{

return (String.Format("{0} + {1}i", real, imaginary));

}

}

In above example I have extended the implementation of the sample Complex class given under operator overloading section. This class has one overridden method named ToString, which overrides the default implementation of the standard ToString method to support the correct string conversion of a complex number.

Hide   Copy Code

Complex num1 = new Complex(5, 7);

Complex num2 = new Complex(3, 8);

*// Add two Complex numbers using the*

*// overloaded plus operator*

Complex sum = num1 + num2;

*// Print the numbers and the sum*

*// using the overriden ToString method*

Console.WriteLine("({0}) + ({1}) = {2}", num1, num2, sum);

Console.ReadLine();

Dynamic polymorphism implemented by **abstract classes** and **virtual functions**.

using System;

namespace PolymorphismApplication {

class Shape {

protected int width, height;

public Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

public virtual int area() {

Console.WriteLine("Parent class area :");

return 0;

}

}

class Rectangle: Shape {

public Rectangle( int a = 0, int b = 0): base(a, b) {

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* height);

}

}

class Triangle: Shape {

public Triangle(int a = 0, int b = 0): base(a, b) {

}

public override int area() {

Console.WriteLine("Triangle class area :");

return (width \* height / 2);

}

}

class Caller {

public void CallArea(Shape sh) {

int a;

a = sh.area();

Console.WriteLine("Area: {0}", a);

}

}

class Tester {

static void Main(string[] args) {

Caller c = new Caller();

Rectangle r = new Rectangle(10, 7);

Triangle t = new Triangle(10, 5);

c.CallArea(r);

c.CallArea(t);

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result −

Rectangle class area:

Area: 70

Triangle class area:

Area: 25

# **Polymorphism via Interfaces**

## **Problem**

You need to implement polymorphic functionality on a set of existing classes. These classes already inherit from a base class (other than Object), thus preventing the addition of polymorphic functionality through an abstract or concrete base class.

In a second situation, you need to add polymorphic functionality to a structure. Abstract or concrete classes cannot be used to add polymorphic functionality to a structure.

## **Solution**

Implement polymorphism using an interface instead of an abstract or concrete base class. The code shown here defines two different classes that inherit from ArrayList:

public class InventoryItems : ArrayList

{

// ...

}

public class Personnel : ArrayList

{

// ...

}

We want to add the ability to print from either of these two objects polymorphically. To do this, an interface called IPrint is added to define aPrint method to be implemented in a class:

public interface IPrint

{

void Print( );

}

Implementing the IPrint interface on the InventoryItems and Personnelclasses gives us the following code:

public class InventoryItems : ArrayList, IPrint

{

public void Print( )

{

foreach (object obj in this)

{

Console.WriteLine("Inventory Item: " + obj);

}

}

}

public class Personnel : ArrayList, IPrint

{

public void Print( )

{

foreach (object obj in this)

{

Console.WriteLine("Person: " + obj);

}

}

}

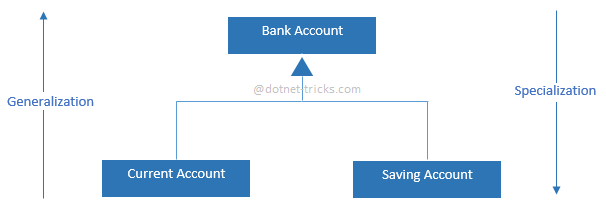
# **Generalization and Specialization**

The process of extracting common characteristics from two or more classes and combining them into a generalized superclass is called Generalization. The common characteristics can be attributes or methods. Generalization is represented by a triangle followed by a line.

https://dotnettricks.blob.core.windows.net/img/oops/generalizationarrow.png

Specialization is the reverse process of Generalization means creating new subclasses from an existing class.

Let’s take an example of Bank Account; A Bank Account is of two types – Current Account and Saving Account. Current Account and Saving Account inherits the common/ generalized properties like Account Number, Account Balance, etc. from a Bank Account and also have their own specialized properties like interest rate, etc.



### Abstraction and Generalization

While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities which perform similar functions with a single construct. Generalization is the broadening of application to encompass a larger domain of objects of the same or different type. Programming languages provide generalization through variables, parameterization, generics and polymorphism. It places the emphasis on the similarities between objects. Thus, it helps to manage complexity by collecting individuals into groups and providing a representative which can be used to specify any individual of the group.

Abstraction and generalization are often used together. Abstracts are generalized through parameterization to provide greater utility. In parameterization, one or more parts of an entity are replaced with a name which is new to the entity. The name is used as a parameter. When the parameterized abstract is invoked, it is invoked with a binding of the parameter to an argument.

### Inheritance and specification

Just like abstraction is closely related with generalization, the inheritance is closely related with specialization. It is important to discuss those two concepts together with generalization to better understand and to reduce the complexity.

One of the most important relationships among objects in the real world is specialization, which can be described as the “**is-a**” relationship. When we say that a dog is a mammal, we mean that the dog is a specialized kind of mammal. It has all the characteristics of any mammal (it bears live young, nurses with milk, has hair), but it specializes these characteristics to the familiar characteristics of canis domesticus. A cat is also a mammal. As such, we expect it to share certain characteristics with the dog that are generalized in Mammal, but to differ in those characteristics that are specialized in cats.

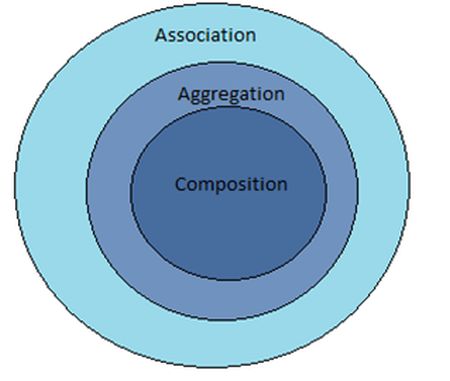
The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization coin: Mammal generalizes what is common between dogs and cats, and dogs and cats specialize mammals to their own specific subtypes.

Similarly, as an example you can say that both IOException and SecurityException are of type Exception. They have all characteristics and behaviors of an Exception, That mean the IOException is a specialized kind of Exception. A SecurityException is also an Exception. As such, we expect it to share certain characteristic with IOException that are generalized in Exception, but to differ in those characteristics that are specialized in SecurityExceptions. In other words, Exception generalizes the shared characteristics of both IOException and SecurityException, while IOException and SecurityException specialize with their characteristics and behaviors.

In OOP, the specialization relationship is implemented using the principle called inheritance. This is the most common and most natural and widely accepted way of implement this relationship.

Class Relationship

In order to modularize/ define the functionality of a one class, that class can uses functions or properties exposed by another class in many different ways. According to Object Oriented Programming there are several techniques classes can use to link with each other. Those techniques are named association, aggregation, and composition.



Company

Composition🡪 Name ("Part of")

Aggregation🡪 Employee (\*the\*)

Association🡪 vendor companies (\*a\*)

# Composition

**Object composition**

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **object composition**.

Broadly speaking, object composition models a “has-a” relationship between two objects. A car “has-a” transmission. Your computer “has-a” CPU. You “have-a” heart. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

**Composition**

To qualify as a **composition**, an object and a part must have the following relationship:

* The part (member) is part of the object (class)
* The part (member) can only belong to one object (class) at a time
* The part (member) has its existence managed by the object (class)
* The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person’s body and a heart. Let’s examine these in more detail.

Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person’s body. The part in a composition can only be part of one object at a time. A heart that is part of one person’s body can not be part of someone else’s body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part’s lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person’s body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a “death relationship”.

And finally, the part doesn’t know about the existence of the whole. Your heart operates blissfully unaware that it is part of a larger structure. We call this a **unidirectional** relationship, because the body knows about the heart, but not the other way around.

The parts of a composition can be singular or multiplicative -- for example, a heart is a singular part of the body, but a body contains 10 fingers (which could be modeled as an array).

In general, if you can design a class using composition, you should design a class using composition. Classes designed using composition are straightforward, flexible, and robust (in that they clean up after themselves nicely).

**Variants on the composition theme**

Although most compositions directly create their parts when the composition is created and directly destroy their parts when the composition is destroyed, there are some variations of composition that bend these rules a bit.

For example:

* A composition may defer creation of some parts until they are needed. For example, a string class may not create a dynamic array of characters until the user assigns the string some data to hold.
* A composition may opt to use a part that has been given to it as input rather than create the part itself.
* A composition may delegate destruction of its parts to some other object (e.g. to a garbage collection routine).

The key point here is that the composition should manage its parts without the user of the composition needing to manage anything.

**Composition and subclasses**

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task well. This makes those classes easier to write and much easier to understand, as they are more focused.
2. Each subclass can be self-contained, which makes them reusable.
3. The parent class can have the subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses. This helps lower the overall complexity of the parent object, because it can delegate tasks to its children, who already know how to do those tasks.

# Aggregation

To qualify as an **aggregation**, a whole object and its parts must have the following relationship:

* The part (member) is part of the object (class)
* The part (member) can belong to more than one object (class) at a time
* The part (member) does not have its existence managed by the object (class)
* The part (member) does not know about the existence of the object (class)

Like a composition, an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship. However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts.

For example, consider the relationship between a person and their home address. In this example, for simplicity, we’ll say every person has an address. However, that address can belong to more than one person at a time: for example, to both you and your roommate or significant other. However, that address isn’t managed by the person -- the address probably existed before the person got there, and will exist after the person is gone. Additionally, a person knows what address they live at, but the addresses don’t know what people live there. Therefore, this is an aggregate relationship.

Similar to a composition, the parts of an aggregation can be singular or multiplicative.

Rule: Implement the simplest relationship type that meets the needs of your program, not what seems right in real-life.

**Summarizing composition and aggregation**

Compositions:

* Typically use normal member variables
* Can use pointer members if the class handles object allocation/deallocation itself
* Responsible for creation/destruction of parts

Aggregations:

* Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
* Not responsible for creating/destroying parts

It is worth noting that the concepts of composition and aggregation are not mutually exclusive, and can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some parts but not others. For example, our Department class could have a name and a Teacher. The name would probably be added to the Department by composition, and would be created and destroyed with the Department. On the other hand, the Teacher would be added to the department by aggregation, and created/destroyed independently.

While aggregations can be extremely useful, they are also potentially more dangerous. Because aggregations do not handle deallocation of their parts, that is left up to an external party to do so. If the external party no longer has a pointer or reference to the abandoned parts, or if it simply forgets to do the cleanup (assuming the class will handle that), then memory will be leaked.

For this reason, compositions should be favored over aggregations.

# Association

To qualify as an **association**, an object and another object must have the following relationship:

* The associated object (member) is otherwise unrelated to the object (class)
* The associated object (member) can belong to more than one object (class) at a time
* The associated object (member) does not have its existence managed by the object (class)
* The associated object (member) may or may not know about the existence of the object (class)

Unlike a composition or aggregation, where the part is a part of the whole object, in an association, the associated object is otherwise unrelated to the object. Just like an aggregation, the associated object can belong to multiple objects simultaneously, and isn’t managed by those objects. However, unlike an aggregation, where the relationship is always unidirectional, in an association, the relationship may be unidirectional or bidirectional (where the two objects are aware of each other).

The relationship between doctors and patients is a great example of an association. The doctor clearly has a relationship with his patients, but conceptually it’s not a part/whole (object composition) relationship. A doctor can see many patients in a day, and a patient can see many doctors (perhaps they want a second opinion, or they are visiting different types of doctors). Neither of the object’s lifespans are tied to the other.

We can say that association models as “uses-a” relationship. The doctor “uses” the patient (to earn income). The patient uses the doctor (for whatever health purposes they need).

Because associations are a broad type of relationship, they can be implemented in many different ways. However, most often, associations are implemented using pointers, where the object points at the associated object.

In general, you should avoid bidirectional associations if a unidirectional one will do, as they add complexity and tend to be harder to write without making errors.

**Reflexive association**

Sometimes objects may have a relationship with other objects of the same type. This is called a **reflexive association**. A good example of a reflexive association is the relationship between a university course and its prerequisites (which are also university courses).

**Associations can be indirect**

In all of the above cases, we’ve used a pointer to directly link objects together. However, in an association, this is not strictly required. Any kind of data that allows you to link two objects together suffices. We can have a Driver class can have a unidirectional association with a Car without actually including a Car pointer member.

In the above example, we have a CarLot holding our cars. The Driver, who needs a car, doesn’t have a pointer to his Car -- instead, he has the ID of the car, which we can use to get the Car from the CarLot when we need it.

**Composition vs aggregation vs association summary**

Here’s a summary table to help you remember the difference between composition, aggregation, and association:

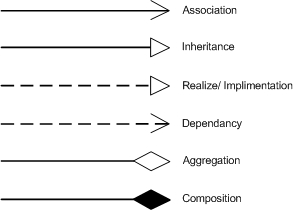
|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Composition** | **Aggregation** | **Association** |
| Relationship type | Whole/part | Whole/part | Otherwise unrelated |
| Members can belong to multiple classes | No | Yes | Yes |
| Members existence managed by class | Yes | No | No |
| Directionality | Unidirectional | Unidirectional | Unidirectional or bidirectional |
| Relationship verb | Part-of | Has-a | Uses-a |

### Class Diagram

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

The Class diagrams, physical data models, along with the system overview diagram are in my opinion the most important diagrams that suite the current day rapid application development requirements.

UML notations:



### Package Diagram

Package diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams provide a visualization of the name-spaces. In my designs, I use the package diagrams to organize classes in to different modules of the system.

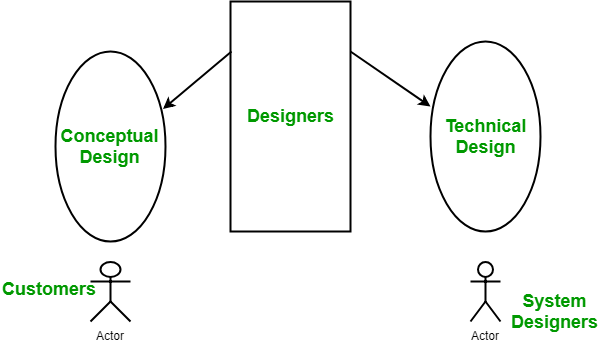
### Sequence Diagram

A sequence diagrams model the flow of logic within a system in a visual manner, it enable both to document and validate your logic, and are used for both analysis and design purposes. Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system.

# **Coupling and Cohesion**

**Introduction:** The purpose of Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS(Software Requirement Specification) document. The output of the design phase is Sofware Design Document (SDD).

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer’s problem.



**Conceptual design of system:**

* Written in simple language i.e. customer understandable language.
* Detail explaination about system characteristics.
* Describes the functionality of the system.
* It is independent of implementation.
* Linked with requirement document.

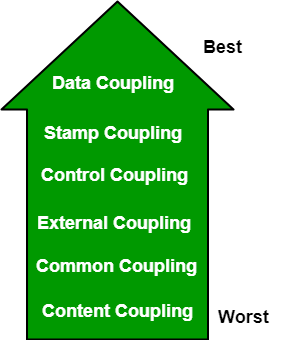
**Technical Design of system:**

* Hardware component and design.
* Functionality and hierarchy of software component.
* Software architecture
* Network architecture
* Data structure and flow of data.
* I/O component of the system.
* Shows interface.

**Modularization:** Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

* Easy to understand the system.
* System maintenance is easy.
* A module can be used many times as their requirements. No need to write it again and again.

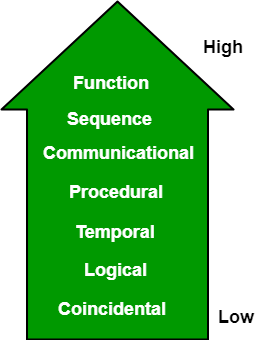
**Coupling:** Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.



**Types of Coupling:**

* **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don’t contain tramp data. Example-customer billing system.
* **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice made by the insightful designer, not a lazy programmer.
* **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
* **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
* **Common Coupling:** The modules have shared data such as global data structures.The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
* **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

**Cohesion:** Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.

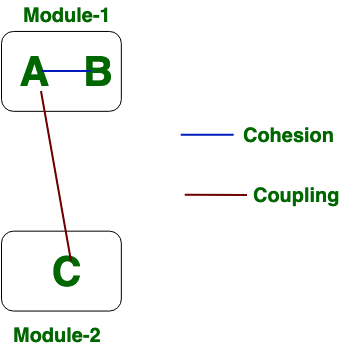


**Types of Cohesion:**

* **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
* **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
* **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record int the database and send it to the printer.
* **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
* **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at init time.
* **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
* **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

**Cohesion:**  
Cohesion is the indication of the relationship within module. It is concept of intra-module. Cohesion has many types but usually highly cohesion is good for software.

**Coupling:**  
Coupling is also the indication of the relationships between modules. It is concept of Inter-module. Coupling has also many types but usually low coupling is good for software.



Now we will see the difference between Cohesion and Coupling. the differences between cohesion and coupling are given below:

|  |  |
| --- | --- |
| **COHESION** | **COUPLING** |
| Cohesion is the concept of intra module. | Coupling is the concept of inter module. |
| Cohesion represents the relationship within module. | Coupling represents the relationships between modules. |
| Increasing in cohesion is good for software. | Increasing in coupling is avoided for software. |
| Cohesion represents the functional strength of modules. | Coupling represents the independence among modules. |
| Highly cohesive gives the best software. | Where as loosely coupling gives the best software. |
| In cohesion, module focuses on the single thing. | In coupling, modules are connected to the other modules. |

**Class members and scope**