Programming Paradigm	Object Oriented Programming	Classes and instances	Information Hiding and Encapsulation	Inheritance – Class Hierarchies
Method binding, Overriding, and Exception	Object-Messages, Inheritance, and Polymorphism	What Is an Interface?	Comparison between C++ and Java	

Programming Paradigm

Paradigm

- A fundamental style of programming. Four Main paradigms are shown.
- A *multi-paradigm programming language* is a programming language that supports more than one programming paradigm.
- The lowest level programming paradigms are machine code (first generation) and assembly language (second generation)

Imperative

- Imperative programming is based on procedural languages. The program is structured following the logic of execution. COBOL, FORTRAN, C, BASIC, PL/1, ALGOL are included in this category. These are also known as high-level Language (third generation).

Declarative

- *In Declarative programming* paradigms, the computer is told what the problem is, not how to solve the problem - the program is structured as a collection of properties to find in the expected result, not as a procedure to follow. Example : the fourth generation language SQL, as well as the family of functional languages and logic programming.

Functional

- *Functional programming* is a subset of declarative programming. Programs written using this paradigm use functions, blocks of code intended to behave like mathematical functions.

Object-oriented

- Later, *object-oriented languages* (like Simula, Smalltalk, C++, C#, Eiffel and Java) were created. In these languages, data, and methods of manipulating the data, are kept as a single unit called an object.

Multi-paradigm languages : C#, which includes imperative and object-oriented paradigms as well as some support for functional programming. F# and Scala, which provides similar functionality to C# but also includes full support for functional programming. Java, C++ also support multi paradigm.

A **fifth-generation programming language** (**5GL**) is a programming language based on problem solving using constraints given to the program, rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some other declarative languages are fifth-generation languages.

Object Oriented Programming

- Object oriented programming is a new way of thinking about computing a problem in computer.
- It is frequently referred to as a new programming paradigm, not a programming language.

To illustrate some of the major features in OOP, let us consider how a system works in a real-world situation and then ask how we could make the computer more closely model the techniques employed in real world situation.

Suppose you wish to send some flowers to your friend (Sunil) on his birthday. He lives in a city many miles away. So you go down to your local florist named Ram, tell him the kinds and number of flowers you want to send, your friend's address and his birthday. He will deliver the flower on that day automatically.

The major steps to fulfill this :

- Find an appropriate agent (Ram)
- Pass a message containing your request to him. It is the responsibility of Ram to satisfy your request.
- Ram is doing a set of operations to fulfill the request. You do not need to know the particular method he will use to satisfy your request. This information is usually hidden from your inspection.

  Ram may deliver a slightly different message to another florist in your friend's city. That florist, in turn, makes the arrangement and passes it, along with yet another message, to a delivery person and so on. In this way your request is finally satisfied by a sequence of requests from one agent to another.

In OOP context step can be :

- You initiate an action by transmission of a message to an agent (object - Ram) responsible for the action.
- The message encodes the request for an action with additional information (arguments) needed to carry out the request.
- If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to the message, the receiver will perform some method to satisfy the request.

If analyze deeply we will notice some important concepts of real-life system :

- Each individual (object) can perform some actions on its own.
- Those actions can be initiated by another object by sending a message and upon completion result is send back.
- In each system there are several objects interacting with each other to achieve a fruitful result.

These concepts of real life system are being directly used in OOP. In OOP

1. Everything is an object
2. Computation is performed by objects communicating with each other, requesting other objects to perform actions.
3. Each object has its own memory
4. Every object is an instance of a class. A class simply represents a grouping of similar objects.

Bundling code into individual software objects provides a number of benefits, including:

1. **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. **Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. **Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program.
4. **Plugging and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Different concepts/properties of OOP

1. Classes and instances (objects)

We observed that some objects are interacting with each other to achieve the objective. Here Ram is one of such objects and belongs to a category (class) named "**Florist**". Some information about florist is common for all florists. Ram being an instance of this category will fit the general pattern. We can use Florist to represent the category (or class) of all florists.

- Here Ram is an entity that has state and behavior and is known as an **object.** Real life object **examples are** chair, bike, marker, pen, table, car etc.

  An object has three characteristics:
  - **state:** represents data (value) of an object.
  - **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
  - **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM or runtime to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Uniquely identified object is also known as instance.

- A class represents a group of objects which have common properties. It defined as a template/blueprint that describes the behavior/state that the objects of its type support. A *class* is the blueprint from which individual objects are created. It is a logical entity. It can't be physical.

- All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.

The following Bicycle class (Java Code) is one possible implementation of a real life bicycle:

```
class Bicycle {
   int cadence = 0;     int speed = 0;      int gear = 1;

   void changeCadence(int newValue) {    cadence = newValue;    }
   void changeGear(int newValue) {       gear = newValue;    }
   void speedUp(int increment) {       speed = speed + increment;      }
   void applyBrakes(int decrement) {       speed = speed - decrement;    }
   void printStates() {
       System.out.println("cadence:" +   cadence + " speed:" +   speed + " gear:" + gear);
   }
}
```

The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

The Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

Java Implementation

```
class BicycleDemo {
   public static void main(String[] args) {
      // Create two different Bicycle objects
      Bicycle bike1 = new Bicycle();        Bicycle bike2 = new Bicycle();

      // Invoke methods on  those objects
      bike1.changeCadence(50);      bike1.speedUp(10);      bike1.changeGear(2);
      bike1.printStates();

      bike2.changeCadence(50);      bike2.speedUp(10);      bike2.changeGear(2);
      bike2.changeCadence(40);      bike2.speedUp(10);      bike2.changeGear(3);
      bike2.printStates();
   }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3

Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming. For each object that you see, ask yourself two questions:

- "What possible states can this object be in?" and
- "What possible behavior can this object perform?"

Real-world objects vary in complexity and one object may contain other objects too.

Software objects are conceptually similar to real-world objects. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

2. Data Hiding, Abstraction, and Encapsulation

OOP follows two important principle :

1. Message is used as vehicle for initiating the activity
2. Information hiding : the client sending the request need not know the actual means by which the request will be honored.

Data Hiding

1. Data Hiding is a feature of OOP languages which doesn't let the program access an object's data directly, keeping it hidden and safe from accidental alteration.
2. The data should not go out directly i.e outside person is not allowed to access the data this is nothing but **"Data Hiding"**.
3. The main advantage of data hiding is we can achieve **security**.
4. By using **private** modifier we can achieve this.

E.g

```
class Employee {
  private int empId;
  private String empName;
  private Date empDOB;
  ...
  ...
}
```

- It is highly recommended to declare data members with private modifier.
- If we hide the data then how it will be available to external world. So we need a way of doing so and that is Encapsulation.

Abstraction and Encapsulation

- Abstraction is a core concept of Computer Science which mainly states that - You don't need to know how the data, pages, words...etc. are being handled to execute the program. You can simply use the functions and write your code by understanding what it the construct does. Only learn and

know what is needed.

- Abstraction and encapsulation both are OOP concepts related to data hiding and they are technically inseparable, they still have their differences in many aspects.

- Abstraction also relates to hiding like encapsulation, but while the former hides complexity, the latter keeps the data it encapsulates by controlling access to them.

Abstraction	**Encapsulation**
• Abstraction is a concept applied in the **design** level. The idea behind abstraction is to focus on **what** rather than **how**. It deals with only the relevant details by hiding the irrelevant ones to reduce complexity thereby increasing efficiency. For example:- Outer layer of a mobile Phone, like it has a display screen and keypad buttons to dial a number. • Abstraction is a data hiding mechanism which highlights only the essential features to make complex programs simpler. It deals with ideas rather than events.	• Encapsulation hides the internal mechanics of **how**. It also hides the details but on the **implementation** level. • It is a method of binding data and codes into a single entity to restrict access from outside world. The idea is to shield the implementation details from external access. It is a method of data hiding. For Example – Inner Implementation detail of Mobile Phone, How keypad button and display screen are connected with each other using circuit.
• It's implemented using **abstract** class and **interface**.	• It's implemented using **protected**, **private**, and **package-private** access modifiers.

Note:

- If we **don't know about implementation** just we have to represent the specification then we should go for interface.
- If we don't know about complete implementation just we have **partial implementation** then we should go for abstract.
- If we know **complete implementation** and if we are ready to provide service then we should go for **concrete class**

Information hiding is also an important aspect of programming in conventional languages. It is generally realized using procedure call. But there are two important distinctions:

1. In OOP, in a message there is a designated receiver (object) for the message. In procedure call there is no designated receiver.
2. In OOP, the interpretation of the message (that is, the method used to respond to the message) is dependent on the receiver and can vary with different receivers. Say in our earlier example, you can give the message to your wife to fulfill the same request. However, the method uses by your

wife to satisfy the request (in all likelihood , simply passing the request on to Ram) will be different from that used by Ram in response to the same request. But in conventional message there can be only one procedure name to accept the call (no duplicate name).

Encapsulation and information hiding are very closely linked concepts. There is a subtle difference between Data Hiding and Encapsulation.

An implementation of Encapsulation :

Encapsulation is putting a security around the data you do want to protect from unauthorized access. It is combining data and methods and allowing the internal data to be accessed by public methods. This can be achieved with access specifiers and public functions.

The implementation of the following class is completely un-encapsulated and hence unsafe (e.g. the account can be changed to be negative).

```
public class BankAccount {
   public int dollars;
}
```

We now have control over how the state is modified, and we can also change the implementation without breaking client code.

However, if we hide the data behind a formally defined interface of methods, we gain flexibility and safety.

```
public class BankAccount {
   private int dollars;

   public void deposit(int dollars) {
      this.dollars += Math.max(0, dollars);
   }
}
```

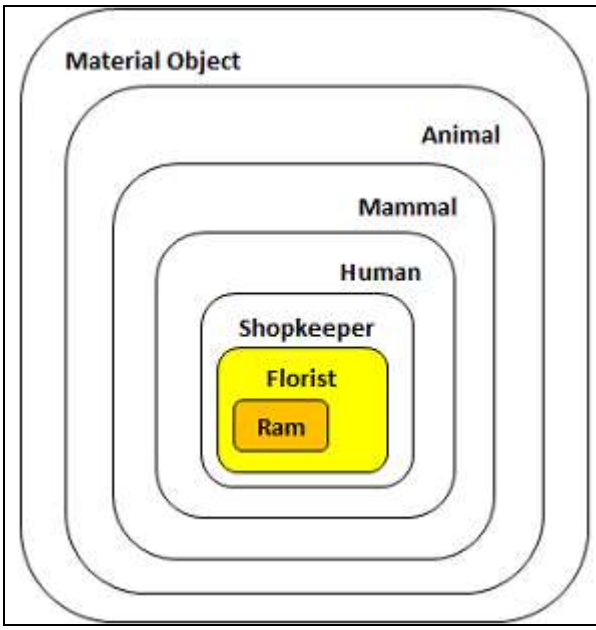
3. Inheritance – Class Hierarchy

If we try to consolidate all the information regarding Ram we will note following points:

- He is florist but he is also a shopkeeper
- He will ask for money for delivering the flowers and in return of payment a receipt will be issued by him. This action is true for grocers, stationers, and other shopkeeper
- Florist is a specialized form of the category Shopkeeper and knowledge of Shopkeeper is also applicable for Florist and hence of Ram.

So we can organize the knowledge of Ram in terms of a hierarchy of categories as shown in figure below.

- The principle that knowledge of a more general category is also applicable to a more specific category is called **inheritance**. Inheritance implements the **IS-A relationship**
- 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**.



- Florist is a specialized form of Shopkeeper
- Furthermore, a Shopkeeper is also a Human (Ram is probably bipedal – walking with two legs)
- A Human is a Mammal (therefore they nurse their young)
- A Mammal is an Animal (it breathes oxygen)
- An Animal is a Material Object (therefore it has mass and weight)

So, quite a lot of knowledge that is applicable to Ram is not directly associated with him, or even with his category Florist.

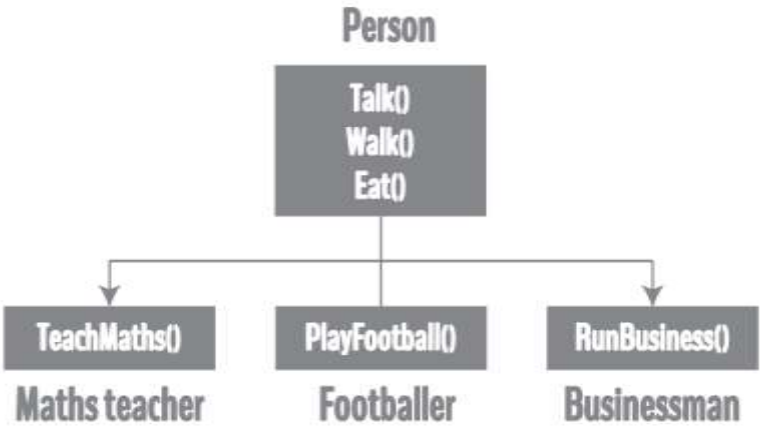
Why inheritance should be used in OOP?

- Suppose, in your game, you want three characters - a **maths teacher**, a **footballer** and a **businessman**.
- Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can **teach maths**, a footballer can **play football** and a businessman can **run a business**.
- Different kinds of objects often have a certain amount in common behavior with each other.

You can individually create three classes who can walk, talk and perform their special skill as shown in the figure below.

In each of the classes, you would be copying the same code for walk and talk for each character.
If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.

It'd be a lot easier if we had a **Person** class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.



Using inheritance now you don't implement the same code for walk and talk for each class. You just need to **inherit** them.

So, for Maths teacher (derived class), you inherit all features of a Person (base class) and add a new feature **TeachMaths**. Likewise, for a footballer and others.

This makes your code cleaner, understandable and extendable.

Advantages	**Disadvantages**
• Reduce code redundancy. • Provides code **reusability**. • Reduces source code size and improves code readability. • Code is more maintainable as it is divided into parent and child classes. • Supports code extensibility by overriding the base class functionality within child classes.	• Base class and child classes are tightly coupled. Hence If you change the code of parent class, it will get affects to the all the child classes. • In class hierarchy many data members remain unused and the memory allocated to them is not utilized. Hence affect performance of your program if you have not implemented inheritance correctly.

4. Polymorphism

- Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms. This means same entity takes multiple forms on different context.

Our universe exhibits many examples of entities that can change form:

- A butterfly morphs from larva to pupa to imago, its adult form.
- On Earth, the normal state of water is liquid, but water changes to a solid when frozen, and to a gas when heated to its boiling point.

This ability to change form is known as polymorphism. It is the ability to take more than one form.

Real World Example of Polymorphism
Example 1

- A Teacher behaves with student.
- A Teacher behaves with his/her seniors.

Here teacher is an object but the attitude is different in different situations.

Example 2

- Person behaves as a SON in house, at the same time that person behaves like an EMPLOYEE in the office.

Polymorphism in OOP refers to the ability of a variable, function or object to take on multiple forms. In a programming language that exhibits polymorphism, objects of classes belonging to the same hierarchical tree (inherited from a common base class) may possess functions bearing the same name, but each having different behaviors.

As an example, assume there is a base class named Animals from which the subclasses Fish and Bird are derived. Also assume that the Animals class has a function named Move, which is inherited by all subclasses mentioned. With polymorphism, each subclass may have its own way of implementing the function. So, for example, when the Move function is called in an object of the Fish class, swimming might be displayed on the screen. In the case of a Bird object, it may be flying.

In effect, polymorphism cuts down the work of the developer because he can now create a sort of general class with all the attributes and behaviors that he envisions for it. When the time comes for the developer to create more specific subclasses with certain unique attributes and behaviors, the developer can simply alter code in the specific portions where the behaviors differ. All other portions of the code can be left as is.

Polymorphism allows the shared code to be tailored to fit the specific circumstances of individual data types. This facilitates incremental development process.

Polymorphism can be implemented in OOP in different ways which we will elaborate later on.

Polymorphism describes a pattern in object-oriented programming in which classes have different functionality while sharing a common interface. Different classes implements common interfaces differently.

Seen in nearly every language in the form of operator overloading. For example in C, we have operators like +, -, *, etc, which operate for all primitive types. In other words we don't have separate "+ int", "+ float", "+ char" etc, operators for each type. The arguments to the + operator may be of a variety of types (the set of primitives). We can say that the + operator is polymorphic.

Reusable software

People have asked for decades why the construction of software cannot mirror more closely the construction of material objects. When we construct a building, a car, or an electronic device, for example, we typically piece together a number of off-the-self components rather than fabricate each new element from scratch. Can software be constructed in the same fashion?

In conventional approach of programming, it is a seldom-achieved goal. A major reason for this is the tight interconnectedness of most software constructed in conventional manner. It is very difficult to

extract from one project elements of software that can be easily used in an unrelated project, because each portion of code typically has interdependencies with all other portions. These interdependencies may be a result of data definitions or may be functional dependencies.

OOP techniques provide a mechanism for cleanly separating responsibilities which results less interconnectedness. Thus by using OOP we can construct large reusable software components.

What Is an Interface?

Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

interface Bicycle {

```
   //  wheel revolutions per minute
   void changeCadence(int newValue);
   void changeGear(int newValue);
   void speedUp(int increment);
   void applyBrakes(int decrement);
}
```

Java Implementation

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {

   int cadence = 0;    int speed = 0;    int gear = 1;

  // The compiler will now require that methods changeCadence, changeGear, speedUp, and
     applyBrakes all be implemented. Compilation will fail if those methods are missing from this class.

   void changeCadence(int newValue) {        cadence = newValue;    }
   void changeGear(int newValue) {        gear = newValue;    }
   void speedUp(int increment) {        speed = speed + increment;      }
   void applyBrakes(int decrement) {        speed = speed - decrement;    }
   void printStates() {
       System.out.println("cadence:" +    cadence + " speed:" +   speed + " gear:" + gear);
   }
}
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is

enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

Comparison between C++ and Java

C++	Java
• You can write C++ program without using classes or objects. • Pointers are available. • Memory allocation and deallocation is the responsibility of the programmer • Has **goto** statement • Automatic casting is available in C++ • Multiple inheritance features available. • Operator overloading is available • Support **three** access specifiers : private, public, protected. • #define, typedef and header files available	• It is purely OOP, not possible to write java program without using at least one class. • No Pointers in Java • Memory allocation and deallocation done by JVM. • No **goto** statement • In some cases, implicit casting is available. Advisable to use casting whenever required. • No multiple inheritance • No Operator overloading • Supports **four** access specifiers: private, public, protected, and package private (default). • #define, typedef and header files are not available, but there are means to achieve them.