**Unit IV**

**OBJECT-ORIENTED LANGUAGES**

**Object-Oriented Languages**:

Concepts of Object-oriented Programming, Inheritances and the type system, Object-oriented features in programming languages.

**Concepts of Object-oriented Programming:**

There are several definitions of what object-oriented programming is. Many people refer to an object-oriented program as any program that deals with entities that may be informally called “objects.” For example, given a stack data structure s, in a procedural language we

would call a push operation to add an element as in:

push (s, x);

Dealing with objects, as we have seen in C++, we tell the stack object to push an element onto itself, as in:

s.push(x);

We have seen that in C++ and Eiffel we can use classes to define and create objects. In particular, object-oriented programming languages are characterized by their support of four facilities:

• abstract data type definitions,

• inheritance,

• inclusion polymorphism, and

• dynamic binding of function calls.

We have already discussed abstract data types extensively. They are used in object-oriented programming to define the properties of classes of objects. Inheritance is a mechanism that allows us to define one abstract data type by *deriving* it from an existing abstract data type. The newly defined type “inherits” the properties of the parent type. Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes. Dynamic binding supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable. The pure terminology of object-oriented languages refers to *objects* that are *instances* of *classes*. An object contains a number of *instance variables* and supports a number of *methods*.

**Classes of objects:**

The first requirement for object-oriented programming is to be able to define abstract data types.

For example,

class stack{

public:

void push(int) {elements[top++] = i;};

int pop() {return elements[--top];};

private:

int elements[100];

int top=0;

};

This class is a particular implementation of a fixed-size stack abstraction.

A client may create as many objects of the stack class as desired:

stack s1, s2;

s1.push(3);

s1.push(4);

s2.push(3);

if (s1.pop() == s2.pop) {...}

Clients may create objects of this class just as they may create variables of language-defined types.

For example, the above fragment creates stacks s1 and s2 as automatic variables. We may also create stacks in the free store:

stack\* sp = new stack;

To access member functions (e.g., pop) the following notations denote equivalent expressions:

(\*sp).pop();

and (more commonly used)

sp -> pop();

While useful, the class facility only addresses the question of how to encapsulate useful abstractions. Inheritance is the mechanism used for this purpose.

**Inheritance:**

Inheritance is a linguistic mechanism that allows us to do just that by defining a new class which “inherits” the properties of a parent class. We may then add new properties to the child class or redefine inherited properties. The terminology in C++ is to *derive* a class from a *base* class. In this case, we want to derive a counting\_stack from stack as shown below:

classcounting\_stack: public stack {

public:

int size(); //return number of elements on the stack

};

This new class simply inherits all the functions of the class stack1. All public member functions of stack become public member functions of counting\_stack. The class stack is called a base class or the parent class of counting\_stack. The class counting\_stack is said to be derived from its base class. The terms *subclass* and *superclass* are also used to refer to derived and base classes respectively.

**Polymorphism:**

Object-oriented languages provide polymorphic variables that may refer to objects of different classes. Object-oriented languages that adopt a strong type system limit the polymorphism of such variables: usually, a variable of class T is allowed to refer to objects of type T or any classes derived from T. As an example, if we have two pointer variables declared:

stack\* sp = new stack;

counting\_stack\* csp = new counting\_stack;

...

sp = csp; //okay

...

csp = sp; //statically not sure if okay--disallowed

The assignment sp = csp; allows a pointer to a class to point to an object of a subclass. That is, the type of the object that is currently pointed at by spcan be of any of its derived types such as counting\_stack. The assignment csp = sp; is not allowed in C++ because C+has a strong type system. If this assignment were allowed, then a later call to csp->size() would be statically valid but may lead to a runtime error because the object pointed at by spmay not support the member function size().

**Dynamic binding of calls to member functions:**

A derived class may not only add to the functionality of its base class, it may also add new private data and *redefine* or *override* some of the operations provided in the base class.

Consider the following example:

stack\* sp = new stack;

counting\_stack\* csp = new counting\_stack;

...

sp.push(); // stack::push

csp.push(); // counting\_stack::push

...

sp = csp; //assignment is okay

...

sp.push();

At run-time spmay be pointing to a stack object or to a counting\_stackobject should the choice of which routine to call be made statically, in which case stack::push() will be called, or dynamically, in which case counting\_stack::push() will be called. Dynamic binding combined with inheritance is a powerful notion. For example, we may define a class polygon and derive various specialized versions of polygons such as square and rectangle. Suppose that polygon defines a function perimeter to compute the perimeter of a general polygon. For example, in Pascal we might implement polygon as a variant record and explicitly call the right perimeter function based on the tag of the variant record. In C, each objectcould contain a pointer to its perimeter function and the call would have to be made indirectly through this pointer. Both of these solutions are not only more verbose but also less secure and maintainable than the solution using inheritance and dynamic binding.

**Inheritance and the type system:**

In this topic we learn about the interaction between inheritance and type consistency rules of the language raises a number of interesting issues, we consider some of these issues.

**Subclasses versus subtypes:**

The concept of subtype with which we defined a new type as a sub range of an existing type. For example, we defined week\_day as a subrange of day. Subtyping introduces a relationship among objects of the subtype and objects of the the parent type such that objects of a subtype may also be viewed as objects of the parent type. For example, a week\_dayis also a day. This relationship is referred to as the *is-a* relationship: week\_day*is-a* day.The subtype relationship is generalizable to user-defind types such as those defined by classes. For example, a counting\_stack*is-a* stack but not vice versa. If a derived class only adds member variables and functions or redefines existing functions in a *compatible* way, then the derived class defines a subtype. Therefore, whether a derived class defines a subtype depends on the definition of the derived class and is not guaranteed by the language.

**Strong typing and polymorphism:**

Strong type systems have the advantage of enabling type errors to be caught at compile-time. Let us assume that we have a base class base and a derived class derived and two objects derived from them:

class base {...};

class derived: public base {...};

...

base\* b;

derived\* d;

We have seen that we may assign d to b but not b to d.

b = d;

will not lead to a type violation at runtime. If substitutability is ensured, the derived type can be viewed as a subtype of the parent type.

***Type extension:***

If the derived class is only allowed to extend the type of the parent class, then substitutability is guaranteed. That is, if derived does not modify any member functions of base and does not hide any of them, then it is guaranteed that any call b.f(...) will be valid whether b is holding a base object or a derived object. Type extension is one of the mechanisms adopted in Ada 95.

***Overriding of member functions:***

In C++, the base class must specify the function perimeter as a virtual function, giving derived classes the opportunity to override its definition.

That is,

class polygon {

public:

polygon (...) {...} //constructor

virtual float perimeter () {...};

...

};

class square: public polygon {

public:

...

float perimeter() {...}; //overrides the definition of perimeter in polygon

};

C++ requires that the signature of the overriding function must be *exactly* the same as the signature of the overridden function.

Now let us consider a member function with a result parameter.

//not C++

class base {

public:

t1 virtual fnc (s1) (...); //s1 is the type of formal paramter;

// t1 is the type of result parameter

...

};

class derived: public base {

public:

t2fnc (s2) (...); //C++ requires that s1 is identical to s2 and t1 is identical to t2

...

};

...

base\* b;

derived\* d;

s1 v1;

s2 v2;

t0 v0;

...

if (...) b = d;

...

v0 = b->fnc(v1); // okay if b is base but what if it is derived?

Again, substitutability means that if b holds a derived object, the call fnc() will work at runtime and a proper result will be returned to be assigned to v0 without any type violations. That means that the result type of the overriding function (t2) must be substitutable to the result type of the overridden function (t1), which must be substitutable to the type of v0, if the last assignement of the fragment is considered to be legal by the compiler.

**Inheritance hierarchies:**

Hierarchical organization is an effective method of controlling complexity. The inheritance relationship imposes a hierarchy and provides a mechanism for the development of a hierarchically organized families of classes.

**Single and multiple inheritance:**

In Simula 67, Ada, and Smalltalk, a new class definition is restricted to have only one base class: a class has at most one parent class. These languages have a *single-inheritance* model. C++ and Eiffel have extended the notion of inheritance to allow a child to be derived from more than one base class. This is called *multiple inheritance*. For example, if we have a class displayable and a class polygon, we might inherit from both to define a displayable rectangle:

class rectangle: public displayable, public polygon {

...

}

The introduction of multiple inheritance into a language raises several issues. For example, there may be name clashes between parents. For example, a displayable class and a bank\_accountclass may both provide a member function draw() and inheriting from both to build a displayable bank\_accountmay cause problems. For example, Eiffel has a construct to **undefined** an inherited feature; it also has a construct to **rename** an inherited feature. The successful use of multiple inheritance requires not only well-designed inheritance hierarchies but also orthogonally designed classes that may be combined without clashing.

**Implementation and interface inheritance:**

Inheritance complicates the issue of encapsulation because the derived classes of a class are a different type of client for the class. On the one hand, they may want to extend the facilities of a parent class and may be able to do so solely by using the public interfaces of the parent class; on the other hand, the facilities they provide to their clients may often be implemented more efficiently if they access the internal representations of their parent classes.

From a software engineering view, interface inheritance is the right methodology but to rely only on interface inheritance requires both a well-designed base class and efficient language implementations. A well-designed inheritance hierarchy is a requirement for the successful use of object-oriented programming in any case. Any hierarchy implies that the nodes closer to the root of the hierarchy affect a larger number of the leaf nodes of the hierarchy. If a node close to the root needs to be modified, all of its children are affected. As a result, even though inheritance supports the incremental creation of software components, it also creates a tightly-dependent set of components. Modifications of base classes may have far reaching impact.

**Object-oriented features in programming languages:**

**Object-oriented programming** (**OOP**) is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) based on the concept of "[objects](https://en.wikipedia.org/wiki/Object_%28computer_science%29)", which may contain [data](https://en.wikipedia.org/wiki/Data), in the form of [fields](https://en.wikipedia.org/wiki/Field_%28computer_science%29), often known as *attributes;* and code, in the form of procedures, often known as [*methods*](https://en.wikipedia.org/wiki/Method_%28computer_science%29)*.* A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "[this](https://en.wikipedia.org/wiki/This_%28computer_programming%29)" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.[[1]](https://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-1)[[2]](https://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-2) There is significant diversity of OOP languages, but the most popular ones are [class-based](https://en.wikipedia.org/wiki/Class-based_programming), meaning that objects are [instances](https://en.wikipedia.org/wiki/Instance_%28computer_science%29) of [classes](https://en.wikipedia.org/wiki/Class_%28computer_science%29), which typically also determine their [type](https://en.wikipedia.org/wiki/Data_type).

Many of the most widely used programming languages (such as C++, Object Pascal, Java, Python etc.) are [multi-paradigm programming languages](https://en.wikipedia.org/wiki/Multi-paradigm_programming_language) that support object-oriented programming to a greater or lesser degree, typically in combination with [imperative](https://en.wikipedia.org/wiki/Imperative_programming), [procedural programming](https://en.wikipedia.org/wiki/Procedural_programming). Significant object-oriented languages include [Java](https://en.wikipedia.org/wiki/Java_%28programming_language%29), [C++](https://en.wikipedia.org/wiki/C%2B%2B), [C#](https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29), [Python](https://en.wikipedia.org/wiki/Python_%28programming_language%29), [PHP](https://en.wikipedia.org/wiki/PHP), [Ruby](https://en.wikipedia.org/wiki/Ruby_%28programming_language%29), [Perl](https://en.wikipedia.org/wiki/Perl), [ObjectPascal](https://en.wikipedia.org/wiki/Object_Pascal), [Objective-C](https://en.wikipedia.org/wiki/Objective-C), [Dart](https://en.wikipedia.org/wiki/Dart_%28programming_language%29), [Swift](https://en.wikipedia.org/wiki/Swift_%28programming_language%29), [Scala](https://en.wikipedia.org/wiki/Scala_%28programming_language%29), [Common Lisp](https://en.wikipedia.org/wiki/Lisp_%28programming_language%29), and [Smalltalk](https://en.wikipedia.org/wiki/Smalltalk).

**Objects and classes:**

Languages that support object-oriented programming typically use [inheritance](https://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29) for code reuse and extensibility in the form of either [classes](https://en.wikipedia.org/wiki/Class-based_programming) or [prototypes](https://en.wikipedia.org/wiki/Prototype-based_programming). Those that use classes support two main concepts:

* [Classes](https://en.wikipedia.org/wiki/Class_%28computer_science%29) – the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods) themselves, i.e. classes contains the data members and member functions
* [Objects](https://en.wikipedia.org/wiki/Object_%28computer_science%29) – instances of classes

Objects sometimes correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".[[7]](https://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-7) Sometimes objects represent more abstract entities, like an object that represents an open file, or an object that provides the service of translating measurements from U.S. customary to metric.

Each object is said to be an [instance](https://en.wikipedia.org/wiki/Instance_%28computer_science%29) of a particular class (for example, an object with its name field set to "Mary" might be an instance of class Employee). Procedures in object-oriented programming are known as [methods](https://en.wikipedia.org/wiki/Method_%28computer_science%29); variables are also known as [fields](https://en.wikipedia.org/wiki/Field_%28computer_science%29), members, attributes, or properties. This leads to the following terms:

* [Class variables](https://en.wikipedia.org/wiki/Class_variable) – belong to the *class as a whole*; there is only one copy of each one
* [Instance variables](https://en.wikipedia.org/wiki/Instance_variable) or attributes – data that belongs to individual *objects*; every object has its own copy of each one
* [Member variables](https://en.wikipedia.org/wiki/Member_variable) – refers to both the class and instance variables that are defined by a particular class
* Class methods – belong to the *class as a whole* and have access only to class variables and inputs from the procedure call
* Instance methods – belong to *individual objects*, and have access to instance variables for the specific object they are called on, inputs, and class variables

Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively [pointers](https://en.wikipedia.org/wiki/Pointer_%28computer_programming%29), serving as actual references to a single instance of said object in memory within a heap or stack. They provide a layer of [abstraction](https://en.wikipedia.org/wiki/Abstraction_%28computer_science%29) which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a [constructor](https://en.wikipedia.org/wiki/Constructor_%28object-oriented_programming%29). A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data.

Object-oriented programming that uses classes is sometimes called [class-based programming](https://en.wikipedia.org/wiki/Class-based_programming), while [prototype-based programming](https://en.wikipedia.org/wiki/Prototype-based_programming) does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of *object* and *instance*.In some languages classes and objects can be composed using other concepts like [traits](https://en.wikipedia.org/wiki/Trait_%28computer_programming%29) and [mixins](https://en.wikipedia.org/wiki/Mixin%22%20%5Co%20%22Mixin).

**Dynamic dispatch/message passing:**

It is the responsibility of the object, not any external code, to select the procedural code to execute in response to a method call, typically by looking up the method at run time in a table associated with the object. This feature is known as [dynamic dispatch](https://en.wikipedia.org/wiki/Dynamic_dispatch), and distinguishes an object from an [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type) (or module), which has a fixed (static) implementation of the operations for all instances. If there are multiple methods that might be run for a given name, it is known as [multiple dispatch](https://en.wikipedia.org/wiki/Multiple_dispatch).

A method call is also known as *message passing*. It is conceptualized as a message (the name of the method and its input parameters) being passed to the object for dispatch.

**Encapsulation:**

Encapsulation is an object-oriented programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of [data hiding](https://en.wikipedia.org/wiki/Information_hiding).

If a class does not allow calling code to access internal object data and permits access through methods only, this is a strong form of abstraction or information hiding known as [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29). Some languages (Java, for example) let classes enforce access restrictions explicitly, for example denoting internal data with the private keyword and designating methods intended for use by code outside the class with the public keyword. Methods may also be designed public, private, or intermediate levels such as protected (which allows access from the same class and its subclasses, but not objects of a different class). In other languages (like Python) this is enforced only by convention (for example, private methods may have names that start with an [underscore](https://en.wikipedia.org/wiki/Underscore)).

Encapsulation prevents external code from being concerned with the internal workings of an object. This facilitates [code refactoring](https://en.wikipedia.org/wiki/Code_refactoring), for example allowing the author of the class to change how objects of that class represent their data internally without changing any external code (as long as "public" method calls work the same way). It also encourages programmers to put all the code that is concerned with a certain set of data in the same class, which organizes it for easy comprehension by other programmers. Encapsulation is a technique that encourages [decoupling](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29).

**Composition, inheritance, and delegation:**

Objects can contain other objects in their instance variables; this is known as [object composition](https://en.wikipedia.org/wiki/Object_composition). For example, an object in the Employee class might contain (point to) an object in the Address class, in addition to its own instance variables like "first\_name" and "position". Object composition is used to represent "has-a" relationships: every employee has an address, so every Employee object has a place to store an Address object.

Languages that support classes almost always support [inheritance](https://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29). This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships. For example, class Employee might inherit from class Person. All the data and methods available to the parent class also appear in the child class with the same names. For example, class Person might define variables "first\_name" and "last\_name" with method "make\_full\_name()". These will also be available in class Employee, which might add the variables "position" and "salary". This technique allows easy re-use of the same procedures and data definitions, in addition to potentially mirroring real-world relationships in an intuitive way. Rather than utilizing database tables and programming subroutines, the developer utilizes objects the user may be more familiar with: objects from their application domain.

Subclasses can override the methods defined by superclasses. [Multiple inheritance](https://en.wikipedia.org/wiki/Multiple_inheritance) is allowed in some languages, though this can make resolving overrides complicated. Some languages have special support for [mixins](https://en.wikipedia.org/wiki/Mixin%22%20%5Co%20%22Mixin), though in any language with multiple inheritance, a mixin is simply a class that does not represent an is-a-type-of relationship. Mixins are typically used to add the same methods to multiple classes. For example, class UnicodeConversionMixin might provide a method unicode\_to\_ascii() when included in class FileReader and class WebPageScraper, which don't share a common parent.

[Abstract classes](https://en.wikipedia.org/wiki/Abstract_class) cannot be instantiated into objects; they exist only for the purpose of inheritance into other "concrete" classes which can be instantiated. In Java, the [final](https://en.wikipedia.org/wiki/Final_%28Java%29) keyword can be used to prevent a class from being subclassed.

The doctrine of [composition over inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance) advocates implementing has-a relationships using composition instead of inheritance. For example, instead of inheriting from class Person, class Employee could give each Employee object an internal Person object, which it then has the opportunity to hide from external code even if class Person has many public attributes or methods. Some languages, like [Go](https://en.wikipedia.org/wiki/Go_%28programming_language%29) do not support inheritance at all.

The "[open/closed principle](https://en.wikipedia.org/wiki/Open/closed_principle)" advocates that classes and functions "should be open for extension, but closed for modification". [Delegation](https://en.wikipedia.org/wiki/Delegation_%28programming%29) is another language feature that can be used as an alternative to inheritance.

**Polymorphism:**

[Subtyping](https://en.wikipedia.org/wiki/Subtyping), a form of [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_%28computer_science%29), is when calling code can be agnostic as to whether an object belongs to a parent class or one of its descendants. For example, a function might call "make\_full\_name()" on an object, which will work whether the object is of class Person or class Employee. This is another type of abstraction which simplifies code external to the class hierarchy and enables strong [separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns).

**Open recursion:**

In languages that support [open recursion](https://en.wikipedia.org/wiki/Open_recursion), object methods can call other methods on the same object (including themselves), typically using a special variable or keyword called this or self. This variable is [*late-bound*](https://en.wikipedia.org/wiki/Name_binding); it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof.

Object-oriented design tries to design a system that consists of objects. Such designs are implemented more easily in object-oriented languages. It is at the design stage that component objects and their relationships are identified. Constructs such as abstract classes that we have seen in Ada, Eiffel, and C++ may be used to document object-oriented designs that can then be implemented in programming languages. The substitutability and proper inheritance properties that we have discussed for programming languages are treated in terms of *is-a* relationship at the analysis and design stages.

In our example, a counting\_stack*is-a* stack and therefore may be substituted anywhere a stack is needed (for example passed to a procedure that expects a stack). But stack is *not* a counting\_stackand therefore a stack may not be substituted for a counting\_stack. A good design rule is to use inheritance to derive a new class when derived\_class*is-a* base\_class. The C++ rule on assignments among derived and base classes may also be defined using the *is-a* relationship. The assignment a = b is allowed if b *is-a*a. While this rule is intuitive and simple to state, it is not always easy to determinewhether two objects are related with the *is-a* relation. For example, we have seen that colorPointis not necessarily a point . Usually, the relationship that holds is “*is-a-kind-of*.” It often takes great care to create *is-a* relationships. In practice, the initial development of the design is not the major problem.

The designer is often able to build an inheritance hierarchy that fits the problem at hand. The problems occur later when the software is extended to meetnew requirements. The difficulties arise when new classes needed to be defined introducing new is-a relationships that are not compatible with previous such relationships. If the inheritance tree needs to be modified significantly, then the impact on the rest of the software can be significant.