3.1 Object –Oriented Programming Paradigms

The object-oriented approach to programming is an easy way to master the management and complexity in developing software systems that take advantage of the strengths of data abstraction. Data-driven methods of programming provide a disciplined approach to the problems of data abstraction, resulting in the development of object-based languages that support only data abstraction. These object-based languages do not support the features of the object-oriented paradigm, such as inheritance or polymorphism. Depending on the object features supported, there are two categories of object languages:

1. Object-Based Programming Languages
2. Object-Oriented Programming Languages

Object-based programming languages support encapsulation and object identity (unique property to differentiate it from other objects) without supporting important features of OOP languages such as polymorphism, inheritance, and message based communication, although these features may be emulated to some extent. Ada, C, and Haskell are three examples of typical object-based programming languages.

Object-based language = Encapsulation + Object Identity

Object-oriented languages incorporate all the features of object-based programming languages, along with inheritance and polymorphism. Therefore, an object-oriented programming language is defined by the following statement:

Object-oriented language = Object-based features + Inheritance + Polymorphism

Object-oriented programming languages for projects of any size use *modules* to represent the physical building blocks of these languages. A module is a logical grouping of related declarations, such as objects Or procedures, and replaces the traditional concept of *subprograms* that existed in earlier languages.

The following are important features in object-oriented programming and design:

1. Improvement over the structured programming paradigm.
2. Emphasis on data rather than algorithms.
3. Procedural abstraction is complemented by data abstraction.
4. Data and associated operations are unified, grouping objects with common attributes, operations, and semantics.

Programs are designed around the data on which it is being operated, rather than the operations themselves. Decomposition, rather than being algorithmic, is data-centric. Clear understanding of classes and objects are essential for learning object-oriented development. The concepts of classes and objects help in the understanding of object model and realizing its importance in solving complex problems.

Object-oriented technology is built upon *object models*. An *Object* is anything having crisply defined conceptual boundaries. Book, pen, train, employee, student, machine, etc., are examples of objects. But the Entities that do not have crisply defined boundaries are not objects. Beauty, river, sky, etc., are not objects.

Model is the description of a specific view of a real-world *problem domain* showing those aspects, which are considered to be important to the observer (user) of the problem domain. Object-oriented programming language directly influences the way in which we view the world. It uses the programming paradigm to address the problems in everyday life. It addresses thue solution closer to the problem domain.

Object model is defined by means of classes and objects. The development of programs using object model is known as object-oriented development. To learn object-oriented programming concepts, it is very important to view the problem from the user's perspective and model the solution using object model.

3.2 Classes and Objected.

The concepts of object-oriented technology must be represented in object-oriented programming languages. Only then, complex problems can be solved in the same manner as they are solved in real-world situations. OOP languages use classes and objects for representing the concepts of abstraction and encapsulation. The mapping of abstraction to a program is shown in Fig. 1.

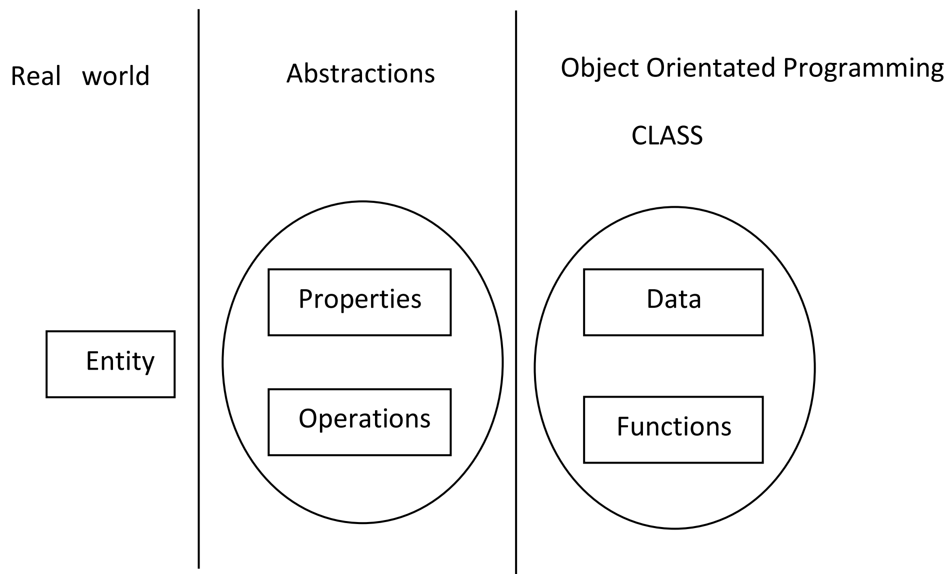

Fig. 1 mapping real world entity to object oriented programming

The *software structure* that supports *data abstraction* is known as *class*. A class is a *data type* capturing the essence of an abstraction. It is characterized by a number of features. The class is a *prototype* or *blue print* or *model* that defines different features. A feature may be a data or an operation. Data are represented by *instance variables* or *data variables* in a class. The operations are also known as behaviors, or methods, or functions. They are represented by member functions of a class in C++ and methods in Java and C#.

A class is a data type and hence it cannot be directly manipulated. It describes a set of objects. For example, *apple is a fruit* implies that apple is an example of fruit. The term "fruit" is a type of food and apple is an instance of fruit. Likewise, a class is a type of data (data type) and object is an instance of class. Similarly *car* represents a *class* (a model of vehicle) and there are a number of instances of car. Each instance of car

is an object and the class car does not physically mean a car. An object is also known as *class variable* because it is created by the class data type. Actually, each object in an object-oriented system corresponds to a *real-world thing*, which may be a person, or a product, or an entity. The differences between class and object are given in Table 1.1.

Table 1.1 Comparisons of Class and Object

Class	Object
Class is a data type.	Object is an instance of class data type.
It generates object.	It gives life to a class.
It is the prototype or model.	It is a container for storing its features.
Does not occupy memory location.	It occupies memory location.
It cannot be manipulated because it is not available in the memory.	It can be manipulated.

Instantiation of an object is defined as the process of creating an object of a particular class.

An object has:

1. States or properties.
2. Operations.
4. Identity.

Properties maintain the internal state of an object. Operations provide the appropriate functionality to the object. Identity differentiates one object from the other. Object name is used to identify the object. Hence, object name itself is an identity. Sometimes, the object name is mixed with a property to differentiate two objects. For example, differentiation of two similar types of cars, say MARUTI 800 may be differentiated by colors. If colors are also same, the registration number is used. Unique identity is important and hence the property reflecting unique identity must be used in an object.

3.3 FEATURES OF OBJECT-ORIENTED PROGRAMMING

The fundamental features of object-oriented programming are as follows:
1. Encapsulation
2. Data Abstraction
3. Inheritance
4. Polymorphism

3.3.1 Encapsulation

The process, or mechanism, by which you combine code and the data it manipulates into a single unit, is commonly referred to as *encapsulation*. Encapsulation provides a layer of security around manipulated data, protecting it from external interference and misuse. In Java, this is supported by *classes* and *objects*.

3.3.2 Data Abstraction

Real-world objects are very complex and it is very difficult to capture the complete details. Hence, OOP uses the concepts of abstraction and encapsulation. Abstraction is a design technique that focuses on the essential attributes and behavior. It is a named collection of essential attributes and behavior relevant to programming a given entity for a specific problem domain, relative to the perspective of the user.

A simple view of an object is a combination of properties and behavior. The method name with arguments represents the interface of an object. The interface is used to interact with the outside world.

Object-oriented programming is a packaging technology. Objects encapsulate data and behavior hiding the details of implementation. The concept of implementation hiding is also known as *information hiding*. Since data is important, the users cannot access this data directly. Only the interfaces (methods) can access or modify the encapsulated data. Thus, *data hiding* is also achieved. The restriction of access to data within an object to only those methods defined by the object's class is known as encapsulation. Also, implementation is independently done improving software reuse concept. Interface encapsulates knowledge about the object. Encapsulation is an abstract concept. Show below gives a clear picture about the different concepts.

3.3.2.1 Comparison of Abstraction and Encapsulation

Abstraction

1. Abstraction separates interface and implementation.
2. User knows only the interfaces of the object and how to use them according to abstraction. Thus, it provides access to a specific part of data.
3. Abstraction gives the coherent picture of what the user wants to know. The degree of relatedness of an encapsulated unit is defined as cohesion. High cohesion is achieved by means of good abstraction.
4. Abstraction is defined as a data type called class which separates interface from implementation.
5. The ability to encapsulate and isolate design from execution information is known as abstraction.

Encapsulation

1. Encapsulation groups related concepts into one item.
2. Encapsulation hides data and the user cannot access the same directly (data hiding).
3. Coupling means dependency. Good systems have low coupling. Encapsulation results in lesser dependencies of one object on other objects in a system that has low coupling. Low coupling may be achieved by designing a good encapsulation.
4. Encapsulation packages data and functionality and hides the implementation details (information hiding).
5. Encapsulation is a concept embedded in abstraction.

Classes and objects represent abstractions in OOP languages. Class is a common representation with definite attributes and operations having a unique name. Class can be viewed as a user-defined data type.

Is a declaration of variables in C. This statement conveys to the compiler that *year* and *mark* are instances of integer data type. Likewise, in OOP, a class is a data type. A variable of a class data type is known as an object. An object is defined as an instance of a class. For example, if *Book* is a defined class,

Book c Book, java Book ;

Declares the variables c Book and java Book of the Book class type. Thus, classes are software prototypes for objects. Creation of a class variable or an object is known as *instantiation* (creation of an instance of a class). The objects must be allocated in memory. Classes cannot be allocated in memory.

3.4 Inheritance

Inheritance allows the extension and reuse of existing code, without having to repeat or rewrite the code from scratch. Inheritance involves the creation of new classes, also called *derived* classes, from existing classes (*base* classes). Allowing the creation of new classes enables the existence of a hierarchy of classes that simulates the class and subclass concept of the real world. The new derived class inherits the members of the base class and also adds its own. For example, a banking system would expect to have customers, of which we keep information such as name, address, etc. A subclass of customer could be customers who are students, where not only we keep their name and address, but we also track the educational institution they are enrolled in. Inheritance is mostly useful for two programming strategies: extension and specialization. Extension uses inheritance to develop new classes from existing ones by adding new features. Specialization makes use of inheritance to refine the behavior of a general class.

3.5 Multiple Inheritance

When a class is derived through inheriting one or more base classes, it is being supported by *multiple inheritance*. Instances of classes using multiple inheritance have instance variables for each of the inherited base classes. However, which to some extent appears like a realization of multiple inheritance.

3.6 Polymorphism

Polymorphism allows an object to be processed differently by data types and or data classes. More precisely, it is the ability for different objects to respond to the same message in different ways. It allows a single name or operator to be associated with different operations, depending on the type of data it has passed, and gives the ability to redefine a *method* within a *derived class*.