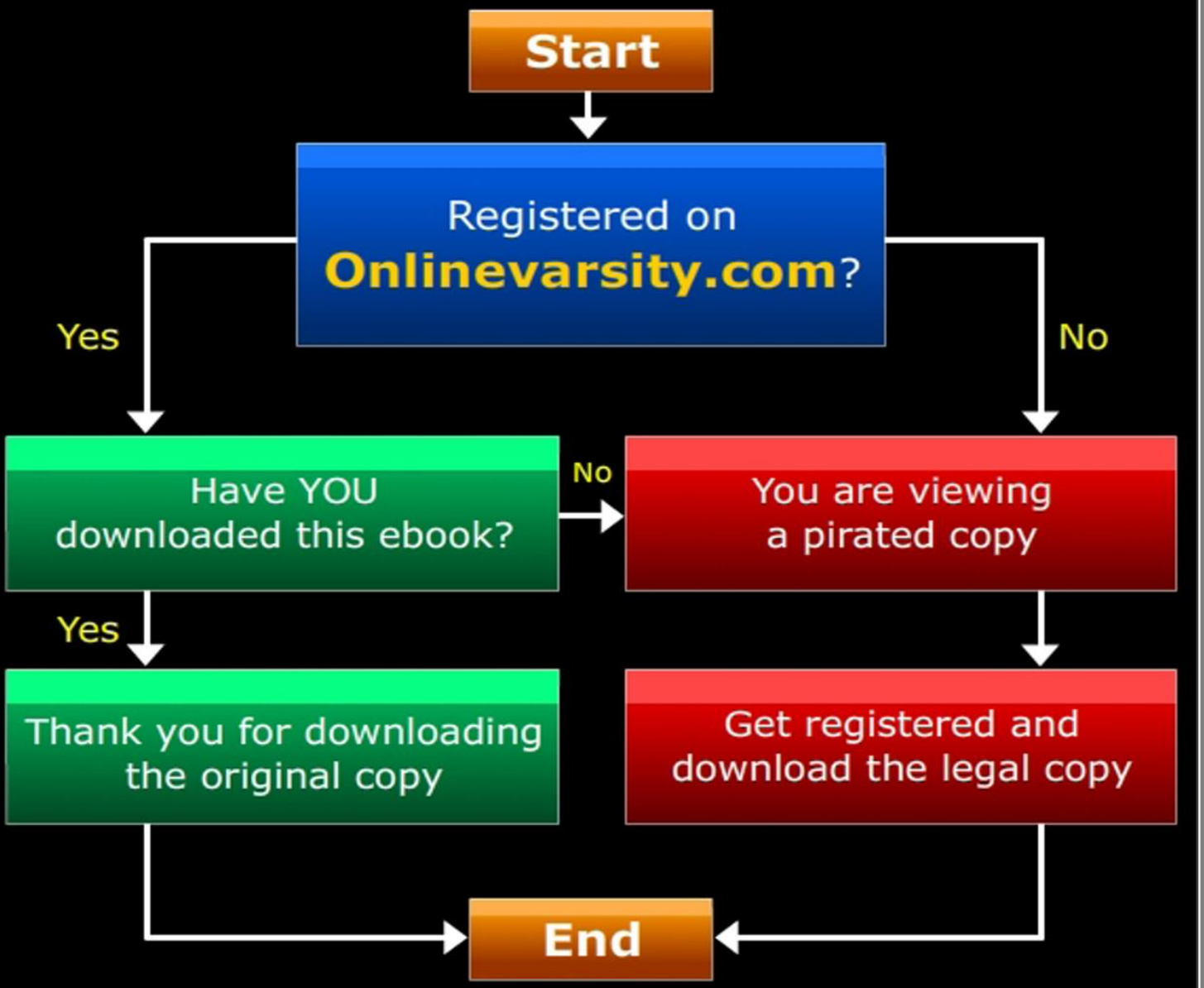
Object-Oriented Programming Concepts



Onlinevarsity
your e-way to learning

Object-Oriented Programming Concepts

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Second Edition - 2012

Aptech™
COMPUTER EDUCATION
Unleash your potential

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

  Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

  The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of Web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

  The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

- Evaluation of instructional process and instructional materials

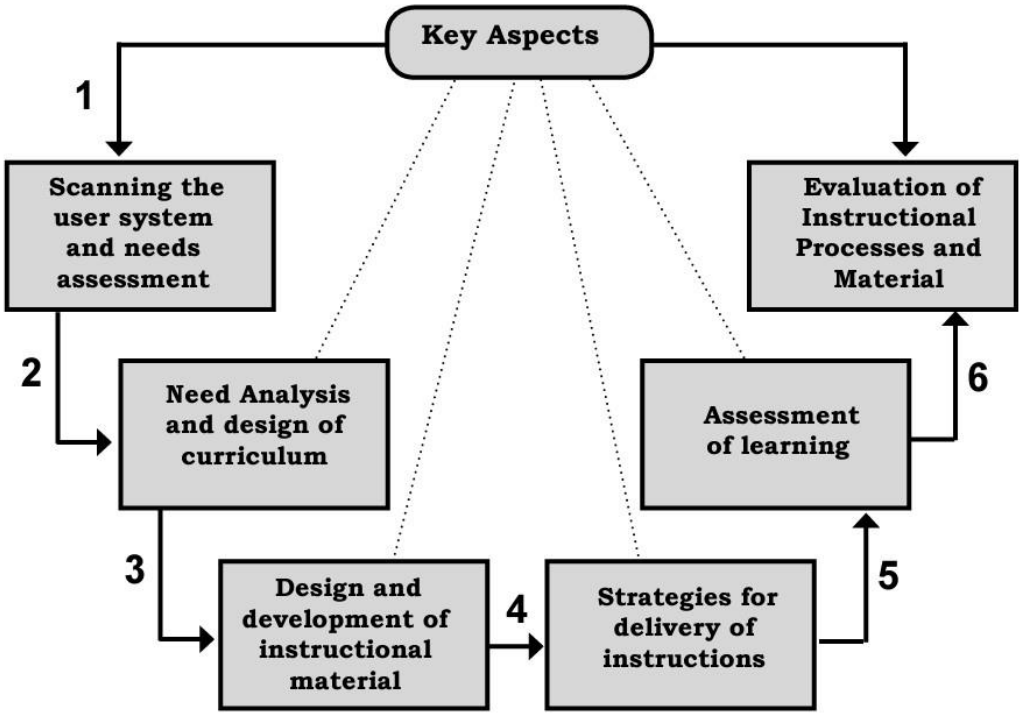
  The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, and ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech™
COMPUTER EDUCATION
Unleash your potential

Aptech New Products Design Model



Aptech™
COMPUTER EDUCATION
Unleash your potential

Preface

Since its inception, object-oriented programming has come a long way and has become a powerful programming methodology. Today, it is established as a basis for programming for almost all modern programming languages such as C++, C#, and Java. It has affected every aspect of the IT industry. The advent of object-oriented programming has made development of high quality and sophisticated software easy and rapid.

This book is targeted for students who have some knowledge of programming in any of the conventional programming languages such as C. The book covers the basic concepts of object-oriented programming. It begins with an introduction to the concept of object-oriented programming, object-oriented design, and some basic software concepts such as class, object and method. Then, it proceeds to explain the fundamental concepts of object-oriented programming such as encapsulation, abstraction, inheritance, and polymorphism. It also describes the various types of inheritance and ways of implementing polymorphism. Every programming concept is explained with a sample program in C++ or C#. Finally, it explains the concept of Frameworks, Reflection, and the use of various types of patterns in programming.

The knowledge and information in this book is the result of the concentrated effort of the Design Team, which is continuously striving to bring to you the latest, the best and the most relevant subject matter in Information Technology. As a part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends and learner requirements.

We will be glad to receive your suggestions.

Design Team

Aptech™ COMPUTER EDUCATION Unleash your potential

Table of Contents

Sessions

Session 1

Introduction to Object-oriented Programming

Objectives

At the end of this session, the student will be able to:

- *Define Object-oriented Programming (OOP)*
- *Differentiate between Object-oriented and Object-based programming*
- *Explain the concepts of OOP*
- *List the advantages and disadvantages of OOP*

1.1 Introduction

Object-oriented programming has gained immense popularity in the recent years since its inception in the 1960s. From software developers to students, everyone is striving to experience object-oriented programming. OOP has revolutionized the entire software industry and has emerged as a totally new concept unlike anything that has come up earlier in programming.

This session is an introduction to OOP and explains the concept of OOP and its features. The session also differentiates OOP with another programming paradigm called Object-based programming. Further, it lists the advantages and disadvantages of OOP.

1.2 OOP – A New Paradigm in Programming

The world is filled with people of different cultures and communities. These people are dependent on a particular language that is used as a medium of communication in their society. There are a varied number of languages existing in the world. Some of them are very difficult to learn, whereas some are easy. The easier and more flexible languages become the common means of communication between people of different communities and hence, more acceptable. This shows that the language that one speaks helps him/her to interact with others. This is applicable not only to natural languages but also artificial languages that are used in computer programming. Since a computer is a machine, it cannot understand natural or human language. So, it is required to have a way to convert natural language into a form which is understandable by the machine. This is not an easy task. It requires lot of intelligence, talent, and logic to make use of available tools and technologies that help you to interact with the computer.

Until today, you might have used procedural programming languages such as C, which is considered as a traditional programming methodology. Procedural programming is relatively simple as it allows code reuse and needs less memory for execution. It uses procedures, also known as methods or functions that contain a series of instructions to be executed. However, one of the drawbacks of procedural programming is that data is not secure.

Concepts

It is exposed to the entire program and easily accessible from anywhere within or outside the program. In procedural programming, it is difficult to derive new data types, which reduces extensibility. Also, it gives more importance to processing of data rather than data itself.

'Object', as a concept, was introduced in the 1960s in a programming language called Simula 67. With Simula came the concept of class, instance, and automatic garbage collection as a part of programming paradigm. The term Object-oriented programming was introduced in 1970s with the invention of a language called Smalltalk by Alan Kay. OOP was coined to represent the use of messages and objects for programming. The popularity of OOP really caught momentum in the 1980s. Several books, journals, and articles have appeared on this subject. OOP has been greeted with even more excitement and enthusiasm than its predecessors such as 'structured programming'.

What is an object? An object is any person or a thing, living or non-living which has some characteristics or attributes which help to describe it. Suppose that a person is holding a cup of tea. The tea cup is an object, the tea inside the cup is also an object, and the tray on which the cup is placed is an object too. You can differentiate between tea and coffee, even though both are drinks, because both have different characteristics in terms of look and taste. Thus, one can characterize both tea and coffee as a type of drink which is their common characteristic. However, the color and taste of each can be used to differentiate them. Also, one can state that tea and coffee are objects belonging to category or class, drink. The color and taste attributes are applicable to any drink and thus, common to both tea as well as coffee. OOP recognizes the need of building an application that represents the real world. This can be achieved using objects. Figure 1.1 shows examples of objects in real world.

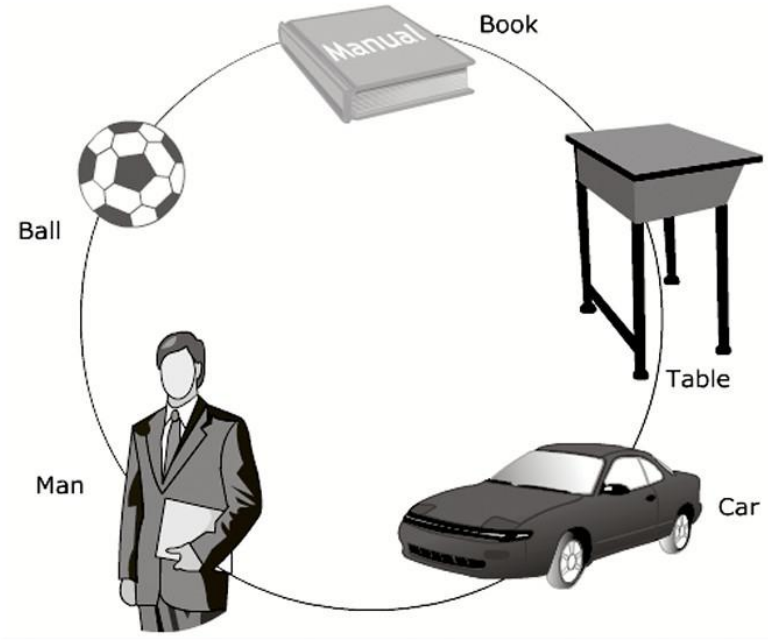


Figure 1.1: Real World Objects

Figure 1.1 shows various real world objects such as ball, book, table, car, and man. Among them, man is a living object, while all others are non-living objects. It is easy to identify each one of them due to their different characteristics. For example, a table has a different shape than a ball. The knowledge of the shape and dimension of each object is already known so that it helps to differentiate between them. This is the basic essence behind OOP.

OOP is often referred to as a new paradigm in programming. A paradigm is a set of concepts and practices that constitute a way of viewing reality for the community sharing them.

OOP makes use of 'objects', which are data structures consisting of attributes and behavior along with their interactions, for designing computer programs. OOP is designed to provide flexibility and maintainability to a program. Due to its strong emphasis on modular design, code becomes simpler and easy to understand as well as maintain.

Code written in non-OOP languages tends to group statements into procedures or functions, each of them performing a specific task. Such a design leads to data becoming more 'global', that is, accessible anywhere in the program. Also, with increase in the size of the program, modification in data through any function would mean that bugs may have wide spread effect on the code and the data. Maintenance of such large sized programs also becomes difficult.

OOP however, allows a programmer to place the data such that it is not directly accessible by the other parts of the program. Instead, the data can be accessed by calling specialized functions or methods that are bundled with data or inherited from other class. In an object-oriented program, several different types of objects can be created. Each type corresponds to a specific kind of data or some real world entity such as 'bank account', 'accountant', 'employee', 'book', and so forth. Also, a program may contain more than one copy of an object that corresponds to each such real world entity. Thus, for the real world entity, `Bank`, a class named `Bank`, and objects of type `Bank` may be created. Each copy of `Bank` object may use the same methods for reading and modifying data but data inside each object will differ from the other `Bank` objects. Figure 1.2 shows two `Bank` objects, `bank1` and `bank2` holding different values for name and address attributes of the two different banks.

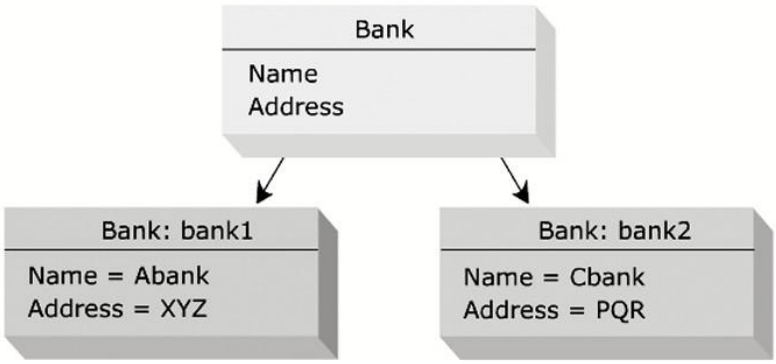


Figure 1.2: Class and Objects

Concepts

1.2.1 OOP Versus Object-based Programming

OOP uses a collection of objects that interact with each other to accomplish a task. This is in contrast with the traditional approach in which a program is a collection of functions or a list of steps. Each object is an individual entity and has a specific responsibility.

OOP includes features such as abstraction, encapsulation, inheritance, modularity, and polymorphism. C++, C#, and Java are some examples of OOP languages.

Object-based programming is more or less a limited version of OOP. Here, there is no implicit inheritance, no polymorphism, and only a reduced number of available objects which are typically the Graphical User Interface (GUI) components. Visual Basic is an example of such a programming language.

Object-based programming is applied to prototype-based system, that is, one based on 'prototype' objects which are not instances of any class. Java Script is an example of such prototype-based systems.

1.3 OOP Concepts

All programming languages do not apply every concept of OOP. OOP basically uses four programming concepts.

1.3.1 Encapsulation

In general, 'encapsulate' means 'to enclose something'. In OOP, encapsulation is used to consciously hide information in a small part of a program. It is a feature used to restrict access to some of the data members by objects. In other words, it provides bundling of data members and methods into an enclosed structure.

Consider a real world situation where a person, Robin wishes to put his money and other valuables in such a place where no one except him can access it. Robin goes to a bank where he is provided with a safe whose key only he can possess. Robin puts his belongings into the safe, takes the key, and goes away. Another person named Chris comes to the bank with the same purpose. However, Chris is allocated another safe into which he can place his valuables. Both Robin's and Chris' safes are isolated from each other and neither can access the other person's safe. Thus, both Robin and Chris accessed the same service of the bank, yet they were only allowed to access items for which they were authorized. The bank safe worked as an enclosing entity which protected access to Robin's valuables from Chris and vice versa. In a similar manner, OOP achieves encapsulation by creating a class which encloses the data members and methods and prevents their unauthorized access.

Concepts

Figure 1.3 shows how the data members and methods can be encapsulated.

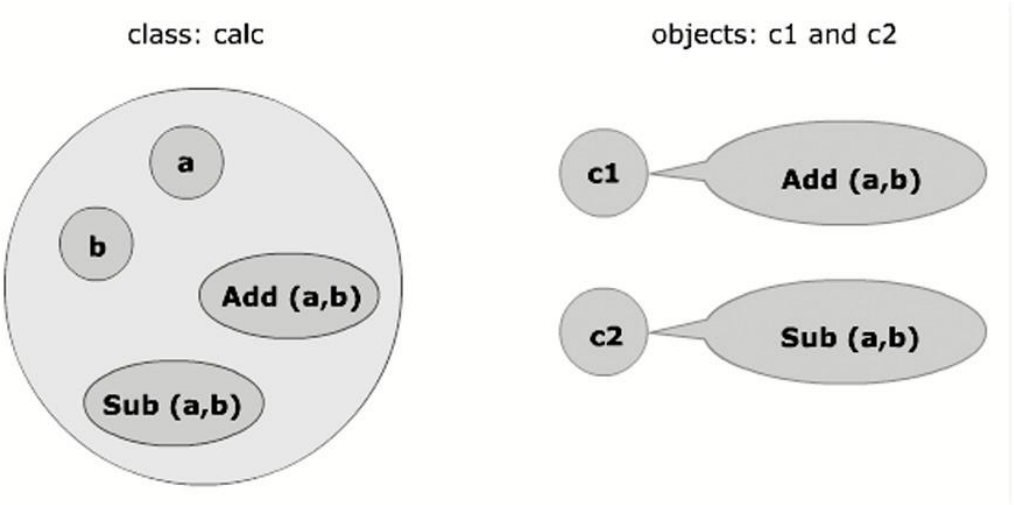


Figure 1.3: Encapsulation

Figure 1.3 shows that the data members of the class, an enclosing structure, are accessible only to the relevant objects that have access to them. Any object that needs access to the data members of class calc must have appropriate permission to access to it.

1.3.2 Abstraction

Abstraction is a mechanism of showing only the relevant details of a process or artifact and hiding the irrelevant details pertaining to an object. For example, you open a book on human anatomy to study the mechanism involved in the walking movement. The first level of anatomy structure will show only the outer skin of the human body. The next level will show the body as composed of bones and muscles. Further, one might check the structure of muscle which contains tissues. The tissue is made of cells, cells again are made of molecules and molecules are made up of atoms. This level of detail might not be necessary and may not be easy to understand.

Thus, any explanation must be given at the right level of detail and the rest should be abstracted. To know how a person can walk by attempting to study the mechanism at the atomic level is not only difficult but irrelevant. It makes it complicated for an average person to understand.

Concepts

Figure 1.4 shows an example of abstraction using the ATM machine.

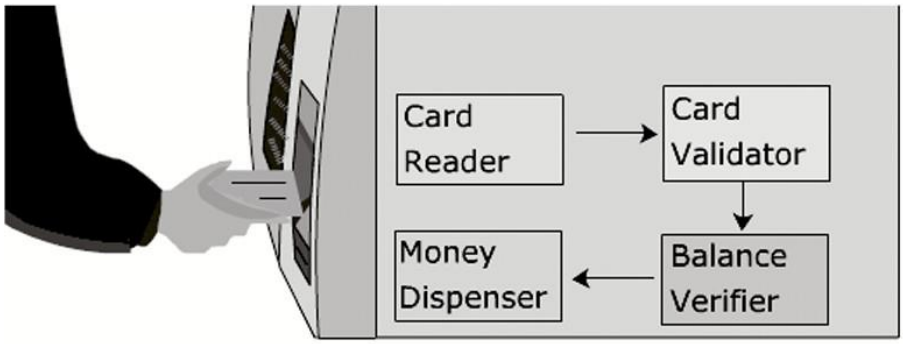


Figure 1.4: Abstraction

Figure 1.4 shows that when a user inserts his/her ATM card in the ATM machine, the transaction is processed and the user receives the money. However, the user is unaware (or abstracted) from the implementation of the process. The user does not need to know how ATM machine accepts the card, how it validates the user credentials, and how cash flows out of the machine. The internal working of the ATM machine is hidden from the user as it is not any of his/her concern. This abstraction of process implementation makes it easier for the user to use the ATM machine.

OOP uses the feature of abstraction to capture only those details of an object that are applicable to a specific scenario, thereby avoiding implementation of unnecessary details.

1.3.3 Inheritance

'Inheritance', in real life, means to pass on characteristics, property, titles, and rights of an individual to his/her successors. In day to day life, one may often find that a person looking similar to his/her father or mother in facial features, height, skin color, eyes, and mannerism. Not only physical looks, but even behavior can be inherited. The way of walking, talking, temperament, and other characteristics are inherited which relates an individual with his/her ancestors. This gives rise to a thought that nature reuses attributes and characteristics of people of one generation to pass on to another generation of the same lineage. However, the descendents of a generation will also have characteristics of their own, apart from those inherited, which helps to differentiate them from their ancestors.

In OOP, inheritance helps to define hierarchical relationships among classes at different levels and gives code reusability. A class can pass on its state and behavior to its child classes. A child class is one which inherits from another class called parent class or super class. There are different types of inheritance namely single, multiple, multilevel, hierarchical, and hybrid inheritance.

Figure 1.5 shows an example of inheritance in real life.

Figure 1.5: Inheritance

Figure 1.5 shows that the features and characteristics of grandfather are inherited by father and passed on to his son. However, each will still have their own specific characteristics that help to differentiate amongst them.

Consider a person named Paul who is a Baker, but a Baker is a Shopkeeper. A Shopkeeper in turn is a Human being, a Human being is a Mammal, a Mammal is an Animal, an Animal is a Living object and a Living object in turn is a Material object.

This hierarchy of inheritance is shown in figure 1.6.

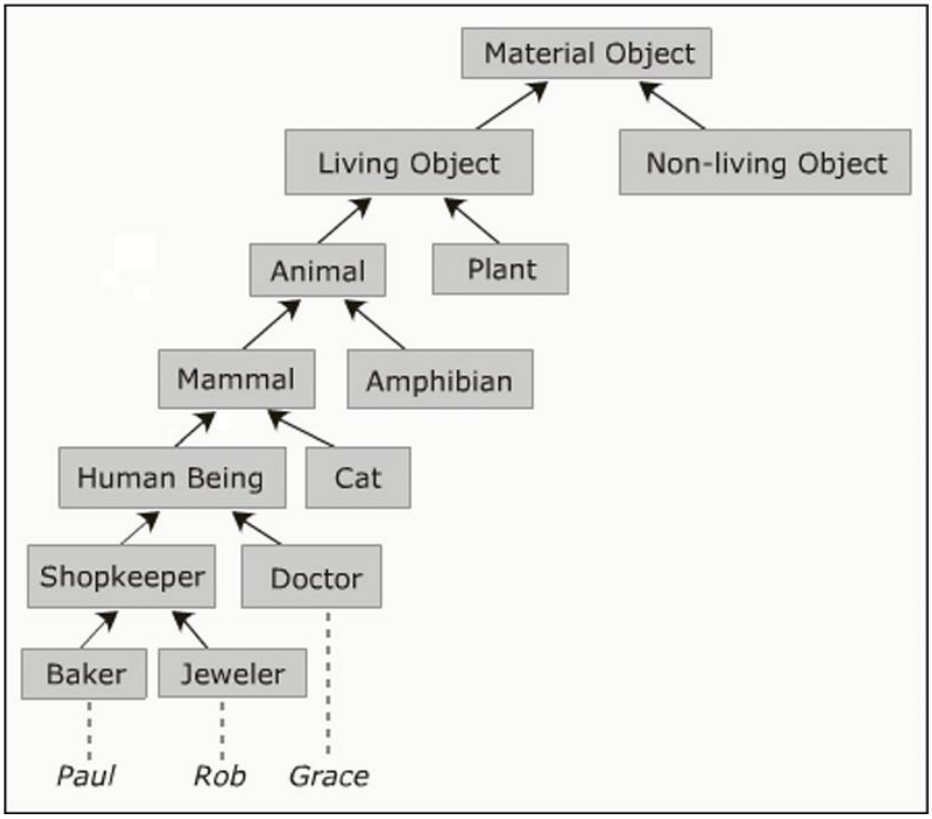


Figure 1.6: Hierarchy of Inheritance

Figure 1.6 shows that each object in the real world can be classified into a category. Based on that category, it will share some characteristics in common with all other objects of that category and also have its own specific attributes to differentiate it from the other objects. Here, Paul is an instance or object of category Baker. However, since Baker is a Shopkeeper, Paul will also have the characteristics of a Shopkeeper which it shares with Jeweler class object, Rob. Similarly, all Shopkeepers are Humans so will share common characteristics with all other objects of category Human Being. This applies to all other objects up in the hierarchy.

1.3.4 Polymorphism

'Poly' means 'many' and 'morphos' means 'forms'. The term polymorphism is a Greek word which means 'many forms'. A polymorph is one that can appear in more than one form. For example, a chameleon changes its color each time to adjust it according to the surroundings. This helps it to hide from its predators. Another example is an Amoeba that keeps changing its shape to facilitate moving around.

In OOP, the term polymorphism means to assign a different usage or meaning to something in different contexts. This allows an entity such a variable, a method, or an object to have more than one form. Polymorphism is implemented in several ways in OOP such as overloading, overriding, polymorphic variable, and generics. Figure 1.7 shows an example of polymorphism in the real world.

Figure 1.7: Polymorphism

Figure 1.7 shows a man in different roles. He can be considered a polymorph, as he behaves differently under different circumstances. Apart from being a human being, he is an employee when he is at work and his duty is to complete the task assigned to him as per his designation in the company. At home, the same person may be a husband and a father whereby his roles and responsibilities change and his duty is to cater to the needs of his family members.

Thus, Encapsulation, Abstraction, Inheritance, and Polymorphism are the four pillars of OOP which make it a robust and flexible tool for designing computer applications.

1.4 Advantages and Disadvantages of OOP

OOP has been widely accepted as a programming methodology. There are a number of reasons why OOP gained popularity and has managed to become a dominant paradigm in programming. However, every coin has two sides. Similarly, there are also many aspects of OOP that have faced resistance from some sections of the programming community. Some advantages and disadvantages of OOP are as follows:

➢ **Advantages:**

- **Code Reusability** – Due to the concept of inheritance, the attributes and behavior of a class are directly inherited by the child class. This enables the child class to use the functionality of the parent class without having to implement the same thing again on its own.

- **Reliability and Flexibility** – OOP system can be more reliable than traditional systems as new functionalities can be created from existing objects. As objects can be accessed dynamically (that is, when required), new objects can be created at any time. Also, OOP provides flexibility that enables the child class to have its own characteristics apart from the inherited ones and also modify the inherited ones.

- **Real World Modeling** – OOP is a system that models the real world better than traditional methods. A user can create objects of classes and associate behavior with them. Thus, OOP is based on objects rather than data and processing.

- **Reduced Code Maintenance** – OOP ensures durability while having lower maintenance costs. As most of the processes of the system are encapsulated, the functions can be reused and modified into new ones without affecting the existing functions. This reusability of code in turn reduces code maintenance.

➢ **Disadvantages:**

- OOP is not a panacea. That is, it is not a solution for everything. OOP is best suited for dynamic and interactive systems, which is evident from its wide acceptance in Computer Aided Design/Computer Aided Manufacturing (CAD/CAM) and engineering systems. However, many information oriented systems such as payroll and accounting, may not benefit extensively from OOP approach.

- OOP is not a technology. It is only an approach to problem solving and not a specific technology.

- OOP has not yet achieved complete acceptance by major vendors. There are still many doubts as to whether OOP will become a major force or fade into oblivion just like the Decision Support System (DSS) which had made great promises, and then faded to obscurity.

- There are not enough trained personnel in OOP. Even though most managers are keen on OOP, yet they do not have time to train their staff in OOP methods. Also, there is a belief that OOP is mainly for graphical workstation systems and there is no urgent need of OOP in mainstream business systems.

- Even though several OOP languages have come up in the market since several years, yet systems written in OOP languages comprise only 1% of systems today.

Aptech COMPUTER EDUCATION Unleash your potential

Session 1

Introduction to Object-oriented Programming

Summary

- OOP is a new paradigm in programming that designs programs by making use of 'objects', which are a copy of real world entities.
- Encapsulation is a feature used to restrict access to some of the data members by objects.
- Abstraction is a mechanism of showing only the relevant details of a process or artifact and hiding the irrelevant details.
- Inheritance helps to define hierarchical relationships among classes at different levels and enables code reusability.
- Polymorphism is a Greek word which means 'many forms' and an object that can appear in different forms is called a polymorph.

Check Your Progress

1. Which of the following options is used to consciously hide information in a part of a program?

A)	Inheritance
B)	Abstraction
C)	Encapsulation
D)	Polymorphism

2. Which of the following options is not an advantage of OOP?

A)	Code reusability
B)	Reliability
C)	Increased maintenance cost
D)	Real world modeling

3. Which of the following options is about OOP are true and which ones are false?

A)	OOP is often referred to as a new paradigm in programming.
B)	OOP system is less reliable than traditional systems.
C)	OOP is a system that cannot model the real world.
D)	C++, C# and Java are some examples of OOP languages.

4. _____ is a feature that describes an object that can appear in more than one form?

A)	Encapsulation
B)	Polymorphism
C)	Abstraction
D)	Inheritance

5. Match the following objects with the corresponding OOP concepts.

Objects		OOP concept	
A)	ATM machine	**1)**	Polymorphism
B)	Car and Vehicle	**2)**	Encapsulation
C)	Chameleon	**3)**	Abstraction
D)	Class and method	**4)**	Inheritance

Session 2

Object-oriented Design

Objectives

At the end of this session, the student will be able to:

- *Explain Object-oriented design (OO Design)*
- *Describe Responsibility-driven Design (RDD)*
- *Explain Agents, Classes, and Instances*
- *Describe Methods, Responsibilities, and Modules*
- *Explain Generalization, Specialization, and Patterns*
- *Explain Coupling and Cohesion*

2.1 Introduction

Working with an Object-oriented language is not in itself a sufficient condition for OOP. Concepts such as Encapsulation, Abstraction, Inheritance, and Polymorphism are the important aspects of OOP that made it so popular. These concepts highlighted OOP as a tool for creating a set of applications which work with coordination of autonomous interacting components, but how can you create such a system?

This session is an introduction to the concepts of OO Design. OO Design is a methodology for planning a system of interacting objects, to solve a problem. When a complex system is created, it becomes difficult to understand the system and its intricacies. OO design helps to explain the complex system by breaking it into smaller units. Each unit will have a specific responsibility that can then be studied individually. This is known as Responsibility-Driven design. Finally, the session explains other software concepts such as Agents, Classes, Methods, Generalization, Specialization, Patterns, Coupling, and Cohesion.

2.2 OO Design

OO Design is a technique that focuses on designing the data and the interfaces to it. It is a process of creating a system where objects coordinate or interact with each other to accomplish a task.

The input for OO Design is provided by Object-oriented analysis in the form of conceptual models, use cases, system sequence diagrams (SSD), user interface documentation, and data models.

Each of these inputs help to describe the system as follows:

- Conceptual model describes the problem domain. It states the user requirements for the system.
- Use case diagram describes the functionality of a system and the users (or actors) that will interact with the system.
- SSD shows a particular scenario of a use case along with the actors involved in the scenario. It shows the sequence in which the actors generate events and the interaction between them.
- User interface documentation describes the look and feel of the product as specified by the user.
- Data model helps to visualize how the data will be represented and used in the system.

The various tasks involved in OO Design are as follows:

- Defining classes and objects from conceptual diagrams to map an entity to a class. This is done using class diagram and object diagram.
- Identifying the attributes of an object.
- Using design patterns to find a solution to a problem. The advantage of using design pattern is that it can be reused to solve a similar problem later.
- Defining the application framework which consists of the set of classes or libraries bundled together so that they can be reused with other applications.
- Define the data model for persistent storage of objects or data.

The end products of OO Design are the system diagrams such as sequence diagrams and class diagrams. These diagrams help to describe the various components of the system and their interactions as follows:

- Sequence diagram is an extension to SSD, to show the specific objects that take part in accomplishing a task. It consists of vertical lines that represent objects and horizontal arrows that represent messages passed between the objects in the order in which they are invoked.
- Class diagram is used to describe the structure of the classes of a system along with its attributes, methods and the relationships between them.

2.3 Responsibility-driven Design

The solution to creating a system of interacting components is a design technique based on delegation of responsibilities and determination of components responsible for executing the responsibilities. This design technique is called Responsibility-driven design (RDD).

Responsibility implies that when a person has been assigned a task, it is his duty to complete it. However, responsibility also implies non-interference or independence to do the task in any way desired. Suppose, a person asks his maid to clean the room, it is the maid's responsibility to carry out the task. Though it is not necessary for the person to watch over her while she completes the task. That is not the nature of responsibility. Instead, having issued the command, it is expected that the desired output will be gained without having to supervise over it. It means that the maid can do the work in whatever way she likes provided she achieves the desired output. This shows an independence to choose the method to complete the task once the responsibility is assigned. RDD helps to assign the responsibilities to the various components of the system as per the specifications given by the user.

In software development, usually the initial specifications for any system are vague and unclear, and contain more general points for every requirement. The first task in RDD is to refine the specifications. This helps to get a better perspective of the final product. The revised specifications can be again verified with the client for agreement. However, it is very likely that over a period of time the specifications may change with the development of the system.

Once the specifications are verified, the design team creates several scenarios to mimic the way a user will interact with the product. As the team walks through the various scenarios, they identify the components that will be part of the scenario and the task each component will perform. Each activity in the scenario has to be assigned to some component as a responsibility. To describe each component and its responsibility, the design team prepares small index cards. Each card contains the name of the software component, the responsibilities and names of other components to which it interacts. Such cards are known as 'Component-Responsibility-Collaborator' (CRC) cards. Figure 2.1 shows the structure of CRC cards.

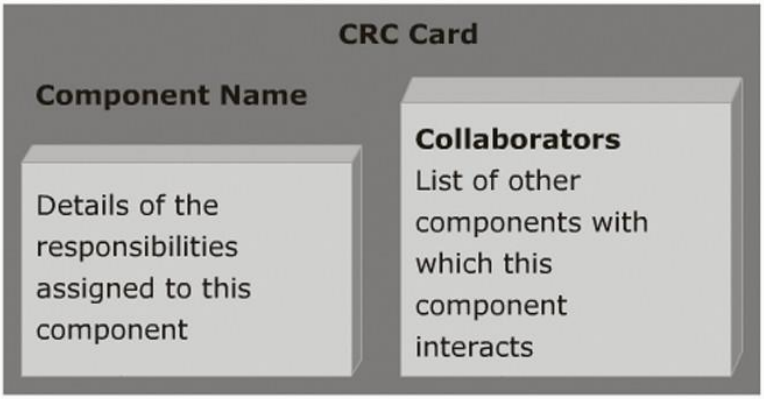


Figure 2.1: CRC Card Structure

Concepts

Once the design team is ready with CRC cards for all components, they begin documentation of the software. This involves creation of user manual and system design documentation. The user manual is intended for the end user and contains details about the usage and interaction with the system. The design documentation contains the details of software development process, decisions taken, and description of the modules or components of the system using system diagrams.

2.4 Software Concepts

Based on RDD, several other software concepts have been defined. These concepts hold true for any system designed using OOP methodology.

2.4.1 Communities of Agents

In real world, often it is seen that a set of people work together to accomplish a project. All the people involved in the project have a dedicated task assigned to them. They work in collaboration with each other by sharing data and documents so as to ensure timely and accurate completion of the task in hand. However, the methodology used to complete the task is left to the discretion and intelligence of each of the project members. Similarly, an object-oriented program is viewed as a community of interacting agents called objects. Each object plays a unique role and provides services to or performs a task for the other members of the community.

Consider an example where a person named Bob wants to send a cake to a friend named Brian. Due to his busy schedule, Bob cannot directly pick up the cake and take it to Brian personally. So, Bob walks to a nearby cake shop, run by a baker named William. Bob tells William the exact details of the kind of cake and the address to which the cake has to be delivered. Now, Bob can be assured that the cake will be delivered directly to Brian. Figure 2.2 depicts the situation.

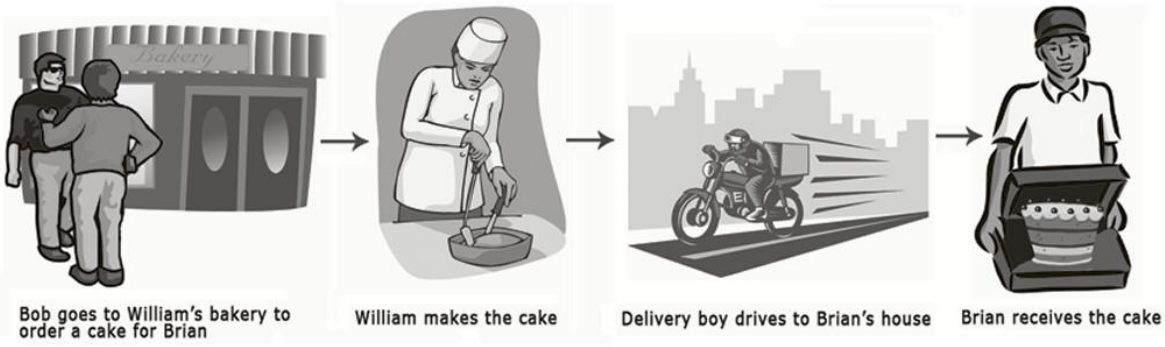


Figure 2.2: The Community of Agents in the Cake Delivery Process

Figure 2.2 shows that the mechanism used to solve the problem was to find an appropriate agent namely, William and to pass a message containing the request. This message consists of the details of the service desired by the requesting object namely, Bob along with data or arguments required to process the request. Now, it is up to the agent William to satisfy the request.

The agent William will use some method or procedure to fulfill the request, which however will not be known to the original requesting agent namely, Bob.

In figure 2.2, William prepares the cake and then gives it to a delivery boy along with the address details of Brian's house. The delivery boy drives to Brian's house according to the address specified and hands over the cake to Brian, the receiver.

The whole situation depicts the role of each individual or agent of a community namely, Bob, William, the delivery boy, and Brian, in accomplishing one task. The request of cake was initialized by Bob, processed by William, and completed by the delivery person. If Bob had decided to deliver the cake himself, it would have cost him a lot of time and energy to complete the task. Instead, he passed on his request to William, who in turn took help of the delivery boy to complete the task. In short, all agents of the community performed their task to complete the cake delivery project.

2.4.2 Classes and Objects

In general terms, a class or a category is a group of people or things having common characteristics. Consider an example, where a person named John buys a house. Now, he asks his young daughter Diana to arrange the various house hold objects. Diana, out of fun, starts arranging things in the house. She puts bed sheet in the kitchen cabinet, utensils in the study room whereas books in the store room and slippers in the rest room. Now, when her mother Kate comes home, she is not able to find any of the utensils such as bowls, plates, knife, and others for cooking. The reason is she obviously looks for them in the kitchen, but as Diana had put them in the study room, Kate could not find it. The point here is Diana, being a kid, did not know where to put what? Meaning she could not categorize the various house hold objects to their appropriate type. That is, spoon is a kitchen utensil hence belongs to the kitchen. Similarly, books belong to the study room. Figure 2.3 shows various objects arranged according to their category or class.

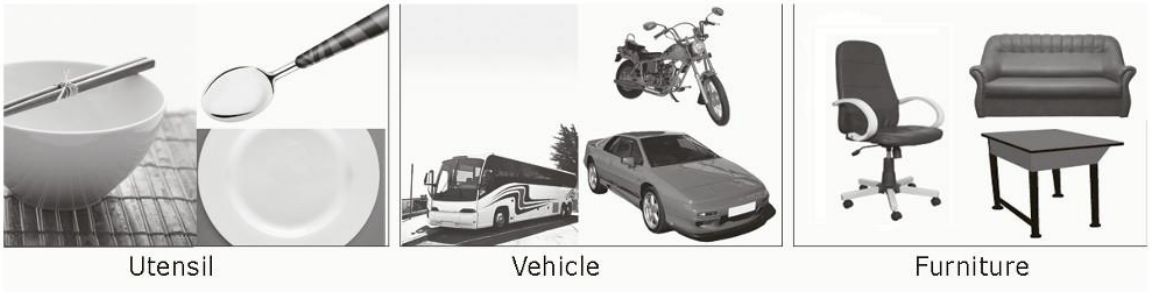


Figure 2.3: Grouping of Objects According to their Category

Figure 2.3 shows grouping of objects such as spoon, plate, and bowl in the category Utensil. Similarly, objects such as bike, car, and bus are grouped under category Vehicle, whereas chair, table, and sofa are grouped under category Furniture.

Concepts

Thus, here Utensil can be called a class and spoon, plate, and bowl can be called objects of class Utensil. These objects have a common characteristic such as a hollow surface that can hold something and they are used in the kitchen for preparing food. Therefore, they have been grouped together in the class Utensil. Similarly, bike, car, and bus can be called objects of class Vehicle because they share common characteristics such as they have tyres, steering, and horn. Also, all vehicles have similar usage that is they are used to drive to a location. Whereas, sofa, chair, and table have common features such as four legs, a flat surface and that they are used to sit. So, they can be classified as objects of class Furniture.

This shows that, grouping objects according to their common characteristics or usage and behavior, simplifies understanding them as well as describing them. OOP also uses this concept of categorizing objects into their particular class.

A class, in OOP, is a structure that defines and encloses the data members or characteristics and methods or behavior pertaining to a particular entity. An object is one copy of the class. Object can also be referred to as an instance of a class. There can be more than one instance of a class in memory at a given time. In figure 2.3, it can be said that spoon belongs to the category Utensil or that spoon is an object of class Utensil.

In the example in figure 2.2, it is clear that William is an instance or object of the class 'Baker' as William is a baker. Similarly, another object of Baker class can be created and a value can be assigned to it. Figure 2.4 shows class Baker with its instances William and Roger.

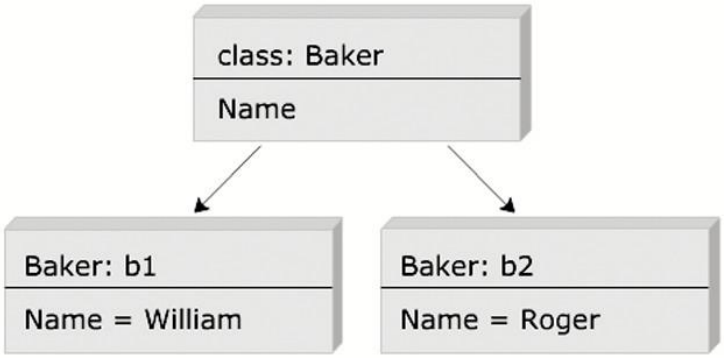


Figure 2.4: Class and Instances

Figure 2.4 shows two objects of class `Baker` namely `b1` and `b2`. The value of `Name` attribute for each object is different. That is, value of `Name` for `b1` is William and that of `b2` is Roger.

2.4.3 Messages and Methods

A method is a set of steps or instructions, or a procedure to accomplish something. Message is a set of words, a phrase or statement to convey something.

From figure 2.2, it is clear that the members of the community interact with each other by making requests. An action is initiated by passing a message to an agent or object responsible for that action. The message contains the request details and additional information or arguments required to fulfill the request. The receiver object, to which message is sent accepts the message and carries out the action indicated in it. This means that in response to a message, the receiver will perform some method to fulfill the request. Figure 2.5 shows how the objects pass messages to each other to fulfill a task.

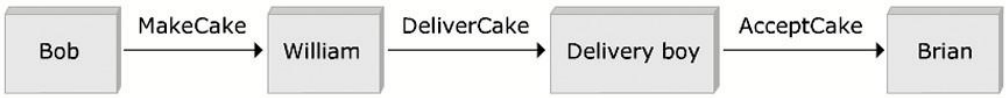


Figure 2.5: Message Passing Amongst Agents in the Cake Delivery Process

Figure 2.5 shows that the objects pass messages to each other to invoke methods belonging to the receiver object to accomplish a task. In the scenario, Bob passes the message 'MakeCake' to William along with the cake details. William, after preparing the cake, passes the message 'DeliverCake' to the delivery boy along with Brian's address. The delivery boy passes the message 'AcceptCake' to Brian along with the cake.

In OOP, objects communicate with each other by invoking methods and passing messages or arguments to the methods to complete a task. The objects can access the methods only if they have appropriate permission for that. The message passing and method invocation can also be better understood from figure 2.6.

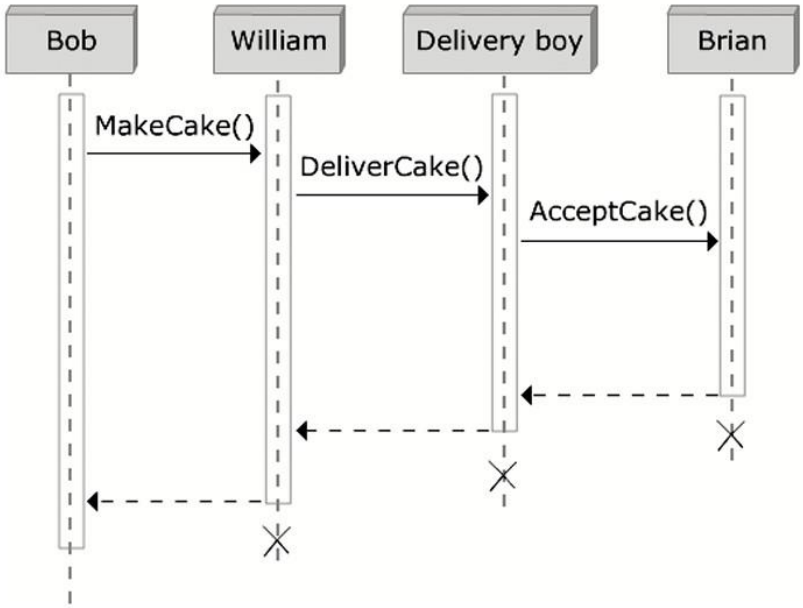


Figure 2.6: Method Invocation Amongst Objects in the Cake Delivery Process

Concepts

Figure 2.6 is a sequence diagram showing the various objects participating in a task. It depicts the sequence in which the various objects are created and how their methods are invoked to complete the scenario. The objects involved are Bob, William, Delivery boy, and Brian. The messages passed or methods invoked are `MakeCake()`, `DeliverCake()`, and `AcceptCake()`. The vertical dashed line represents the life line of an object and the vertical rectangle shows the time for which the object remains alive. Once the task of the object is over, its life time ends which is represented by the symbol 'X'.

2.4.4 Responsibilities

One of the fundamental concepts in OOP is to describe the behavior of objects as responsibilities. In the example in figure 2.2, Bob simply makes a request for cake delivery; however, it is the responsibility of agent William to prepare the cake and deliver it to the receiver agent Brian. William does carry out his responsibility but in a little different manner. He only prepares the cake and then asks the delivery boy to deliver it to Brian. The task is completed when the delivery person hands over the cake to Brian. Each object in the scenario carried out its responsibility with accuracy.

This structure of making a request without having to bother as to how the request will be fulfilled, gives freedom to the individual agents or objects to carry out their task using any technique they desire. That is, there will be no interference from the original requesting object as long as the task is completed as desired. This structure is also known as Chain of Responsibility. Figure 2.7 shows the responsibilities of various agents in the cake delivery process.

Figure 2.7: Responsibilities of Agents in the Cake Delivery Process

2.4.5 Modules

The commonly used method of solving complex problems is to divide the problem into smaller units or modules. Then, analyze each small unit to understand and solve the problem. In the cake delivery scenario shown in figure 2.2, imagine what would have happened if Bob had tried to make the cake himself and then travelled to the Brian's house to deliver the cake to him personally. Bob would have had to spare time from his daily routine to go to the grocery store to buy the necessary material for making a cake. After that he would need additional time to read the recipe of the cake he desired to make. Then, after he mixes the ingredients of the cake as per the instructions in the recipe, he has to wait for the cake to bake. After the cake is ready, he will have to decorate and pack it in a box. Then, he will have up to travel Brian's house.

The scenario shows that since Bob decided to do everything on his own, it cost him a lot of time, energy, and money. Also, he had to take a day off from work and face the inconvenience of travelling to Brian's house. This shows that when a lot of work is assigned to only one object, it becomes very difficult, time consuming, and complicated to complete the task.

However, as per figure 2.2, Bob did not try to make the cake himself nor did he deliver the cake. Instead, he asked William the baker to do the job. William, again finding it inconvenient to deliver the cake himself, simply makes the cake and asks the delivery boy to deliver the cake. Due to this division of work, the task of sending a Cake to Brian was accomplished without Bob having to do anything much by himself, except order the cake. Also, the other objects involved had minimal task to do so that the whole problem got solved in less time and with accuracy. Figure 2.8 shows the approximate time it took Bob to deliver the cake when he does everything himself and figure 2.9 shows the time it took Bob when the task was divided among other agents.

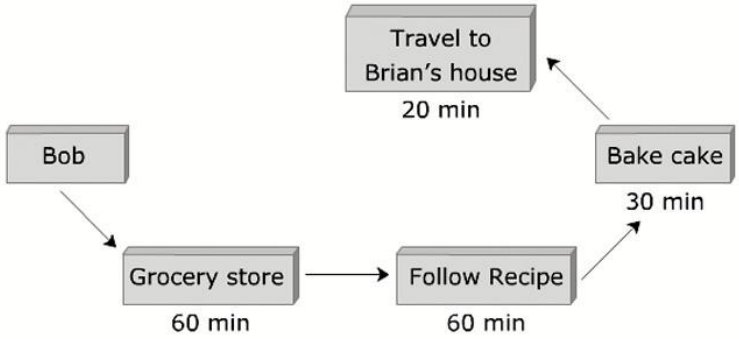


Figure 2.8: Time Taken when Bob does Everything Himself

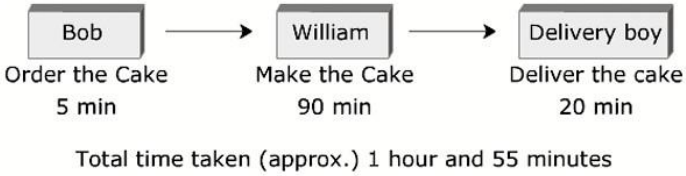


Figure 2.9: Time Taken when the Task is Divided Among Other Agents

Figure 2.8 shows that by dividing the task between multiple people, Bob was able to achieve his goal in almost half the time than he would have done it himself as shown in figure 2.9. Similarly, in OOP, each system can be simplified to the right level of abstraction to understand it better. The study of human anatomy is an example of this. One can achieve better understanding of the complex structure of human body by taking each of the parts, one at a time and studying it. Each part in turn will be considered a module.

Concepts

2.4.6 Generalization and Specialization

In the real world, you may find many objects which are a specialized version of a more general form of the same object. For example, a Baker and a Jeweler both are shopkeepers but a Baker sells cakes whereas a Jeweler sells ornaments. This means that Baker and Jeweler are a more specialized version of a general category of Shopkeepers. Both have the general or common characteristics of a shopkeeper and in addition they also have special characteristics of their own. Figure 2.10 shows the concept of generalization and specialization.

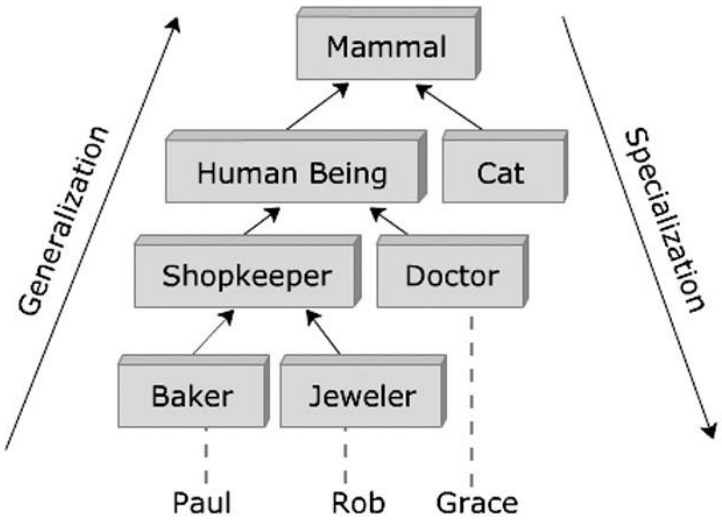


Figure 2.10: Generalization and Specialization

Figure 2.10 shows that if you travel top-down in the inheritance hierarchy, the objects start becoming more specific and the number of sub divisions increases till you reach the last level. Also, the characteristics at each level become more refined and it becomes clear as to the type of object you are talking about. This is known as specialization. In specialization, each object along with the characteristics inherited from its super class, also exhibits its own special features which help to differentiate it from other objects belonging to the same category.

In contrast, when you travel bottom-up in the inheritance hierarchy, the objects start becoming more general and the number of sub-divisions decreases till you reach the top or the root of the hierarchy. Also, characteristics at each level become more common so that you cannot point at any one particular object. Instead, you can make out a set of objects belonging to a particular category. This is known as generalization. In generalization, the specific characteristics of an object are masked and only the common features of the category to which the object belongs are exposed.

In figure 2.10, when one travels from category Mammal down to Baker, you can identify an object Paul who is a baker. However, when you travel from Paul upward to Mammal, you start losing identity of the objects since the categories start to become common such as Baker, Shopkeeper, and Human Being, so that you cannot identify which object of that type is being referred to.

2.4.7 Patterns

A pattern is a shape, model, or structure appearing again and again to create a design. Take an example of the printed tiles used for flooring. The tiles are printed so that they need to be arranged in such a way that the print falls in the right place and a pattern is formed. Once a pattern is formed out of combination of first set of tiles, the rest of the flooring is completed by copying the same pattern all over. The first set of tiles became a pattern or a solution for the rest of the floor.

Similarly, when faced with a new problem, most people will consider referring to a similar problem encountered in the past, to which a solution had been successfully applied. That solution can be used as a model and can be tried out on the new problem by making relevant changes as per the different circumstances. Such a solution to that problem which has been used effectively is described as a pattern.

The design patterns are strategies which are language-independent and used for solving commonly encountered Object-oriented design problems. Design patterns are proven techniques for implementing robust, reusable, and extensible Object-oriented software.

For example, suppose a person is developing a Web-based application in which part of the application will be hosted on one computer and other part will run on other computer linked by a network connection. Creating a direct connection between these computers and transmitting data over this connection are details irrelevant to the actual working of the application. One way to address this issue is to use a type of pattern called proxy. The proxy will be a link which will hide the network connection details. Objects will interact with the proxy and be least concerned about any type of network connection involved in the process. On receiving a request, the proxy will package it and transmit it over the network to the other computer. The response of the other computer will also be received by the proxy and passed back to the client. In this way the client is completely unaware of the details of the network. Figure 2.11 shows the situation of a client making request through a proxy.

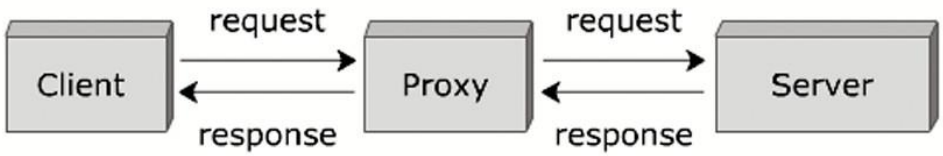


Figure 2.11: Proxy Pattern

Figure 2.11 shows how a proxy hides the network and protocol detail from the client. Due to this pattern, the client is only concerned with sending the request and receiving the response. The client is not concerned as to how the request will be sent on the network.

2.4.8 Coupling and Cohesion

Coupling is the degree to which the components know about each other. Coupling can be loose or tight. Loose coupling means a group of components which can operate independent of each other whereas in tight coupling, the components are largely dependent on each other.

When objects are loosely coupled, the change in one does not affect the other. So, the software becomes easily customizable. The software designers always suggest designing solutions in such a way that objects have minimal dependency on one another.

Suppose there are two components named X and Y. Y knows the details of X because X told Y about it. Or one can say X exposed its details to Y. In this case, the only data that Y has about X is what X has exposed through its interface Therefore, components X and Y are said to be loosely coupled. This type of coupling is highly preferred.

On the other hand, component X has its own characteristics and is responsible for its behavior, no other component can interfere with its functioning. If component X gives access to Y to change its behavior that is if Y gains access to the implementation of X, any change to that implementation in X will break Y. This is called tight coupling. This type of coupling is not preferred because any changes to one component can mean that other components would also have to be altered.

The presence of too much coupling can lead to undesirable situations. So, it is advisable to reduce coupling amongst components since connections between components affects modification, maintenance, and reuse.

While coupling refers to how components interact with each other, cohesion deals with a single component and how focused it is on a specific task. Suppose a person goes to a shop where there is only one attendant. This shop has various objects such as books, stationary, photocopy machine, printer, phone recharge, and many other facilities. Now, there are several customers waiting. One wants to buy a pencil, one wants to get photo copies, one wants a phone recharge, one wants to get a print done and others also have some such demand. As there is only one attendant, he has to do all the work. While he is serving one customer, other customers have to wait even though they do not need the same resource or service as the other customers. In other words, one component is overburdened with too much work which reduces the efficiency and reusability. Such a component is very less cohesive since it is doing too many tasks.

Figure 2.12 shows a less cohesive component.

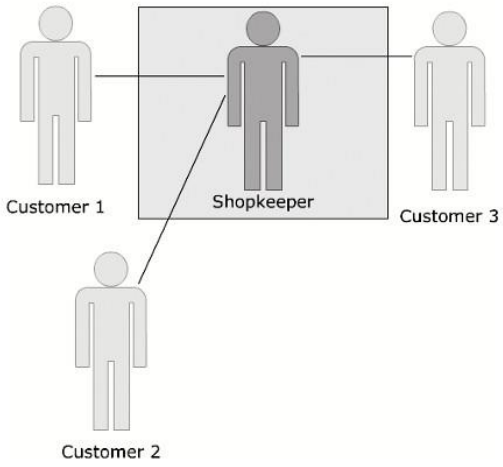


Figure 2.12: Low Cohesion

In the scenario, due to the long waiting time, the customers start to walk away and the shopkeeper starts facing loss. To address this problem, the shopkeeper hired few workers to help him in the shop. Now, when several customers arrive, one worker would do the printing, other would do the photocopying, another would sell the stationary, and the shopkeeper would handle the phone recharge. Thus, when any customer came, based on the demand, the concerned worker or component would serve the customers so that they do not have to wait. As the shop keeper is well-focused on only one purpose, the component is considered cohesive. The more focused a component is on one task, the higher the cohesiveness. That is, instead of one component doing everything, the task is divided among other components, each having a very specific or cohesive role. Figure 2.13 shows a highly cohesive component.

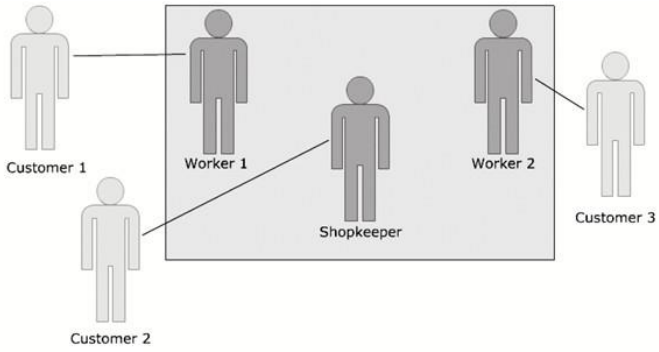


Figure 2.13: High Cohesion

Similarly, in OOP when a specific task is assigned to separate classes, it makes the classes more specialized and reusable to other classes that might need to use the functionality.

Summary

- Responsibility-driven design is a design technique based on delegation of responsibilities and determination of components responsible for executing the responsibilities.
- An object-oriented program is viewed as a community of interacting agents called objects.
- A class is a structure that defines and encloses the data members and methods pertaining to a particular entity. An instance is one copy of the class.
- An action is initiated by passing a message to an agent (or object) responsible for that action. In response to a message, the receiver will perform some method to fulfill the request.
- The commonly used method of solving complex problems is to divide the problem into smaller units called modules.
- A pattern is a solution to a problem which has been used effectively in a similar problem encountered earlier.
- Coupling is the degree to which the components know about each other. Cohesion is the degree to which a particular component is designed for specific and focused purpose.

Check Your Progress

1. Which of the following option is a structure that defines and encloses the data members and methods pertaining to a particular entity?

A)	Object
B)	Class
C)	Agent
D)	Module

2. _____ is a result of dividing a complex problem into smaller units for better understanding.

A)	Class
B)	Method
C)	Object
D)	Module

3. Match the following description with its corresponding concept.

Description		Concept	
A)	Degree to which components know about each other	**1)**	Pattern
B)	A reusable solution to a problem	**2)**	Coupling
C)	Defines behavior of an object	**3)**	Object
D)	An instance of a class	**4)**	Method

4. Which of the following option is about RDD is true and which one is false?

A)	RDD does not help to design a system of interacting agents
B)	RDD helps to assign responsibilities to individual components
C)	RDD helps to make system modular
D)	RDD is not useful to in simplifying a complex system

Check Your Progress

5. Which of the following option is the degree to which a particular component is designed for a specific and focused purpose?

A)	Coupling
B)	Community
C)	Cohesion
D)	Class

Try It Yourself

1. St.Xavier's School is a primary and secondary education school in Chicago. It has a capacity of 2000 students. There are classes for standard I to X. John and Mary study in 10th standard in this school. Subjects such as Mathematics, Science, English, and French are taught in the school. Apart from studies, sports activities such as Cricket, Volley ball, Foot ball, and Tennis are also conducted here. Identify the various objects in the scenario and classify them into appropriate categories or classes.

2. Mr. Mathew lives in Los Angeles. He wishes to send some important documents to his lawyer Mr. Jackson in San Francisco. Mr. Mathew goes to a local courier service office and submits his parcel to the attendant named Larry. Larry registers the parcel and then hands it over to the delivery boy Craig. Craig travels to San Francisco and delivers the courier to the attendant Mac at the courier branch office of San Francisco. Mac registers the courier and hands it to the delivery boy Frank. Frank drives to Mr. Jackson's house and gives him the parcel. Identify the agents in the scenario and draw a diagram to depict RDD.

Session

3 Classes and Methods

Objectives

At the end of this session, the student will be able to:

- *Explain Class*
- *Describe Visibility Modifiers*
- *Explain Methods*
- *Explain Static data fields and Constant data fields*
- *Describe Accessor, Mutator, and Forward declaration*

3.1 Introduction

All Object-oriented programming languages support some or all the features of object-oriented programming (OOP) such as Encapsulation, Abstraction, Inheritance, and Polymorphism. In OOP, each object can be categorized into some class. The characteristics of that object are enclosed within the class as data members. To access and manipulate these data members, the object has to make use of methods which define the behavior of the class.

This session explains the structure of a class and its methods. Further, the session explains about visibility modifiers that are used to restrict access to the class members. Finally, the session describes other members of a class such as Static data fields, Constant data fields, Accessor, Mutator, and Forward declaration.

3.2 Class

Consider a real world entity such as a Horse. A horse is a living thing. It belongs to the category of herbivorous animals. Now, there are several herbivorous animals in the world. So, a question may arise as to what is it about a Horse that differentiates it from other animals? The answer is the characteristics of a Horse, such as its shape, size, look, color, and behavior, help to differentiate it from other animals and also from other animals of its type. So, what are these characteristics? For example, a horse has four feet, hair on its neck, a tail, and hooves on which it walks. The horse has a particular color. It eats grass and it can run fast.

Figure 3.1 shows the entity Horse with its characteristics.

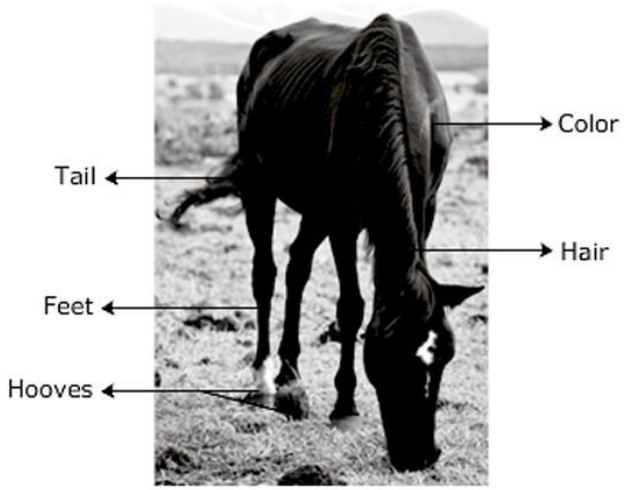


Figure 3.1: Entity and Characteristics

Figure 3.1 shows an entity Horse having the characteristics also called attributes such as Color, Hair, Feet, Hooves, and Tail. Also, it has behavior such as Run, Eat, and Whine.

Now, each horse is bound to have all these characteristics. Hence, one can bundle these characteristics into a category called 'Horse' as shown in figure 3.2.

Horse
Characteristics:
Color
Hair
Feet
Hooves
Tail
Behavior:
Run
Eat
Whine

Figure 3.2: Bundling of Characteristics and Behavior of an Entity

Concepts

Figure 3.2 shows bundling of the characteristics and behavior of entity Horse. This bundling of common attributes and behavior of an entity into an enclosed structure is called a class. Therefore, category Horse can be rephrased as class Horse and its characteristics can be called its attributes as they help to describe the entity Horse. Now, when one wants to talk about a particular horse, one has to say something about its attributes.

For example:

Horse 1 has Color – Brown, Hair – Black, and Tail – Long

Horse 2 has Color – Black, Hair – Grey, and Tail – Short

The example describes two copies of the entity Horse both having different values for the same attributes such as color, hair, and tail. These copies of entity Horse, that is Horse 1 and Horse 2, can also be termed as the objects of class Horse.

Thus, in OOP, a class is a type or a software construct that encloses the data members and functions or behavior of an entity into an enclosed structure. This prevents direct access to these members from outside the class. Each OOP language has its own syntax and terminology for creating a class, but the basic structure of a class remains the same. The syntax of a class is shown here:

Syntax:

```
<access-modifier> class <class-name>
{
      <data-members>
      <methods>
}
```

where,

`access-modifier`: Used to restrict access to the class.

`class-name`: Name of the class.

`data-members`: Attributes declared in the class.

`methods`: Functions declared in the class.

The structure of a class contains a visibility modifier followed by the keyword '`class`' and class name. The two curly braces '`{ }`' mark the start and end of the limit of the class. All data members and methods declared within the curly braces are enclosed inside the class and cannot be directly accessed from outside. Data members are also known as fields, attributes, or variables. They are used to hold data pertaining to that class. A method is a set of instructions to operate on the data members. The C# code given in Code Snippet 1 shows creation of a class.

Code Snippet 1:

```
public class Horse
{
    string color;       // data member
    public void display()     // method
    {
     Console.WriteLine(color);
    }
}
```

In Code Snippet 1 you can identify a class named `Horse` declared in OOP language C# (pronounced as C sharp) along with its data field `color` and method `display()`. The method `display()` is used to print the value of the attribute color. Here, `public` is a visibility modifier which makes the class `Horse` and method `display()` publicly accessible, that is, to all other classes.

A class definition can also be represented as a class diagram. Figure 3.3 shows the structure of a class diagram.

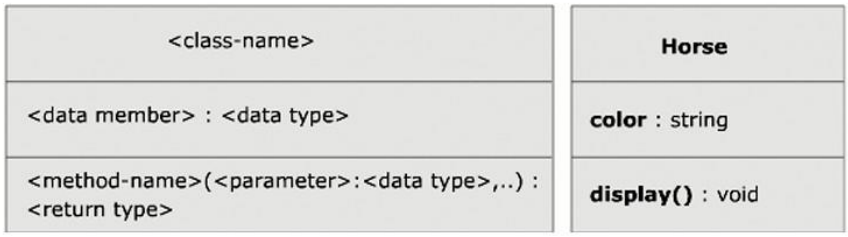


Figure 3.3: Class Diagram

Figure 3.3 shows the diagrammatic representation of the class `Horse` using Class diagram. A Class diagram has three sections. The first section contains the name of the class. The second section has the list of attributes with their names and data types. The third section contains the methods with their parameters and return type. Hence, the first section contains the class name `Horse`, second section has the attribute `color` with its data type `string`, and third section has the method `display()` with its return type `void`.

3.2.1 Visibility Modifiers

The example in figure 3.3 makes use of the keyword public which is a visibility modifier. A visibility modifier is used with class, data fields as well as methods to restrict access to them from outside. All OOP languages provide such visibility modifiers to specify which members of a class are accessible outside and which are not. Even it applies to the class itself, that is, whether a class is exposed to other classes or not. Visibility modifiers are also known as access modifiers.

There are several other visibility modifiers in different languages. However, the three visibility modifiers namely, public, private, and protected are used in all OOP languages. The description of the access modifiers are as follows:

- `public` - A public access modifier states that any type or member declared public is accessible to all other classes within the scope or even outside the scope of the class or package, or assembly in which it is declared. A package or assembly is a mechanism of grouping related classes together as one unit.
- `private` - A private access modifier states that any type or member declared private is accessible only to the code of the same class.
- `protected` - A protected access modifier states that any type or member declared protected is accessible to the code of the same class or to a class derived from that class.

Figure 3.4 shows how class members can be secured using visibility modifiers.

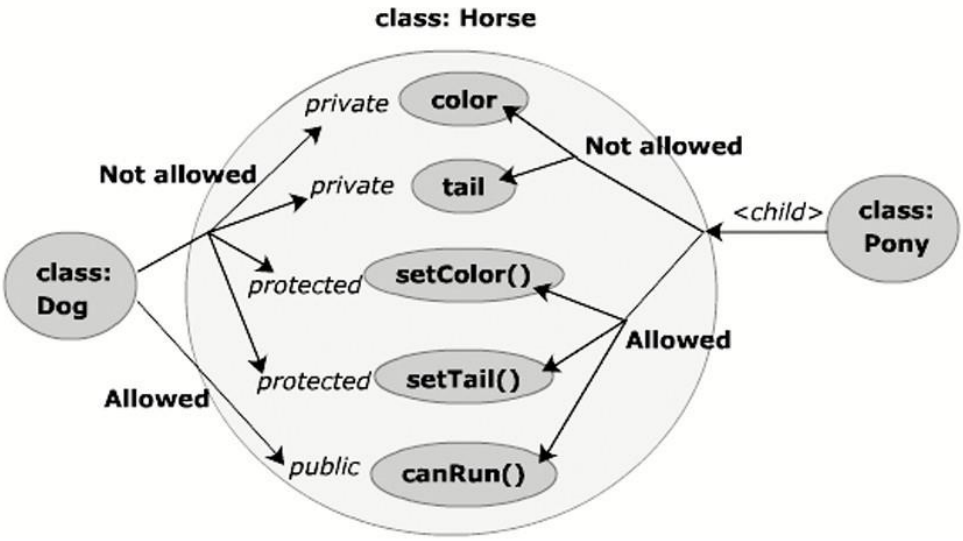


Figure 3.4: Visibility Modifiers

Figure 3.4 shows three classes namely, class `Horse`, class `Pony`, and class `Dog`. The class `Horse` contains two data members namely `color` and `tail`. Both are declared `private`. Also, it has methods namely `setColor()` and `setTail()` which are declared `protected` whereas `canRun()` has been declared `public`. Now, class `Pony` is a child class of `Horse`. Hence, as per rule, it has access to the `public` as well as `protected` members of class `Horse` but no access to the `private` members. Similarly, class `Dog` is an external class and not a child of class `Horse`. Therefore, it has access only to the `public` members but is denied access to the `private` as well as `protected` members of the class `Horse`.

Apart from visibility modifiers, all OOP languages follow some naming conventions and coding conventions. This means that a programmer does not have the liberty to use any case for the name of a class, variable, or method. Also, one cannot give any random name nor use any symbol, or combination of letters of one's choice for a class, variable, or method. A name given to any class, variable, or method is known as an identifier.

Identifier names have some rules or naming conventions which vary with different languages. For example, majority of languages do not allow the use of any special characters except underscore '_' as the first or even intermediate letter for an identifier name. However, Java language allows the symbol '$' as well as '_' in identifier names. Similarly, names of classes, variables, and methods may be in lower case, upper case, or title case in different programming languages. This may even differ with companies and institutions that have their own standards and conventions specific to their company or even to a project. Likewise, the way a method is written, commenting of code, declaring of variables, and such other things also have rules or coding conventions which have to be observed by the programmer while coding.

3.3 Class Members

A class contains several members such as variables, constructor, methods, objects, and other such members.

3.3.1 Methods

Consider the class Horse defined in figure 3.3. The class Horse has an attribute named color. To manipulate this attribute, an object of a class can make use of methods. A method defines the behavior of a class. It consists of the actions that an object or instance of a class can perform on the data members. Similar to a class, a method also has a definite structure or syntax. The basic syntax of a method is shown in the syntax:

Syntax:

```
<access-modifier>  <return-type>  <method-name>  (<data-type>  <parameter-name>,…)
{
    // statement1
    // statement2
    <return-value (or expression)>
}
```

where,

`access-modifier`: States the visibility of the method to others.

`return-type`: Specifies the type of value returned by the method.

`method-name`: Is the name of the method being declared.

`parameter-name`: The name of arguments inside the method declaration.

`data-type`: The data type of the parameter.

`return-value (or expression)`: The value or expression to be returned from the method.

The section within the two curly braces '{}' is known as the method body. All instructions of the method are to be written within the curly braces '{}'. Also, any variable declared within the curly braces of the method, can be accessed only inside the method. That is its scope is limited to the method body and is not recognized anywhere, not even in the same class.

For example, for a class `Horse`, one can define a method to set the value of its attribute `color` as shown in the C# code given in Code Snippet 2.

Code Snippet 2:

```
public string setColor(string col)
{
      return col;
}
```

In Code Snippet 2:

access-modifier is `public`

return-type is `string`

method-name is `setColor`

parameter is `col`

data-type of the parameter is `string`

return-value is `col`

Code Snippet 2 shows the method definition. The method declaration consists of a return value, method name, and argument list. This is also called the method signature. Therefore, in the example of C#, the first line, '`string setColor(string col)`', is known as the signature of method `setColor()`. A method can return a value of primitive type, object type, or `void`. A return type `void` means that the method does not return anything.

Concepts

3.3.2 Object

To access any member of a class, a copy of that class has to be created. This copy is called the object of the class. Therefore, one can create an object of the class `Horse` created in Code Snippet 1 and add the method `setColor()` created in Code Snippet 2 into the class.

Next, one can create an object of the class `Horse` to access the method `setColor()` and set the value of the color attribute as shown in the C# code given in Code Snippet 3.

Code Snippet 3:

```
public class Horse
{
    private string color;       // attribute

    public void setColor(string col)      // method
    {
     color = col;
    }
   static void Main()
   {
     Horse h1 = new Horse();  // creating object of class Horse
     h1.setColor("Black");
   }
}
```

In Code Snippet 3, the statement '`Horse h1 = new Horse()`' is used to create an object `h1` of class `Horse`. The new operator is used to assign memory to the object `h1` of class `Horse`. The statement '`h1.setColor("Black")`' shows how the object `h1` uses the '.' dot operator to access the method `setColor()` and passes the value 'Black' as a parameter. This value will be assigned to the parameter `col` of the method `setColor()` and then the value of `col` will be assigned to the attribute `color` in the statement '`color = col`'.

Similarly, one can create another object `h2` of the `Horse` class and use method `setColor()` to set the value of `color` attribute to 'White'.

This gives rise to the two types of horses. One will be Black and the other will be White as shown in figure 3.5.

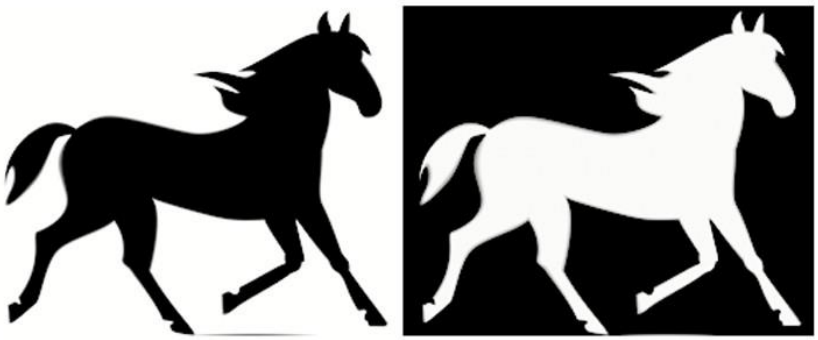


Figure 3.5: Black and White Horses

3.3.3 Constructor

In Code Snippet 3, the statement `h1.setColor()` is used to set the value of the `color` attribute after the object `h1` is created. At times it may be required that as soon as an object of a class is created, its attributes should be assigned some values without having to invoke any method of the class. In this case, it is required to have such a method that gets automatically called when an object of a class is created. OOP has such a method called the constructor.

A constructor is a specialized method that has the same name as the class name. It is used to construct an object of a class and initialize its data members. A class must have at least one constructor. If a constructor is not defined in the class, the compiler will create a default (or no-argument) constructor. A no-argument constructor is one which has zero arguments. Constructor arguments are used to initialize the fields of an object at runtime. A constructor cannot return a value. Constructors cannot be inherited in the child class, but a parent class constructor can be invoked from the constructor of its child class. The basic syntax of creating a default and a parameterized constructor is shown here:

Syntax:

Default or no-argument constructor

```
<constructor-name> ()
{
   // initialization statements
}
```

where,

`constructor-name`: The name of constructor.

`initialization statements`: The statements for initializing the class members.

Parameterized constructor

```
<constructor-name> (<data-type> <parameter-name>,..)
{
   // initialization statements
}
```

where,

`constructor-name`: Name of the constructor.

`data-type`: Data type of the constructor arguments.

`parameter-name`: Name of the constructor argument.

`initialization statements`: Statements for initializing the class members.

The C# code given in Code Snippet 4 shows creation of a default or no-argument constructor.

Code Snippet 4:

```
public class Horse
{
    string color;
     //no-argument constructor
     Horse()
     {
            color = "Brown";     // line1
     }
}
   static void Main()
   {
        Horse h1 = new Horse();
   }
}
```

In Code Snippet 4, `Horse()` is a no-argument constructor. The statement '`color = "Brown"`', is used to assign the value 'Brown' to the data field named `color`. In the statement '`Horse h1 = new Horse()`', `h1` is an object of class `Horse`. The `new` operator is used to assign memory to the object `h1` of class `Horse`. Also, the statement '`new Horse()`', invokes the no-argument constructor `Horse()`. When the constructor is invoked, the value 'Brown' gets assigned to the data field `color` as shown in line 1 in the code, thereby initializing the data field as soon as the object `h1` is created.

Concepts

The C# code given in Code Snippet 5 shows creation of a parameterized constructor.

Code Snippet 5:

```
public class Horse
{
    string color;
     //parameterized constructor
     Horse(string col)
     {
            color = col;
     }
}
  static void Main()
  {
        Horse h1 = new Horse("Black");
    }
}
```

In Code Snippet 5, the statement `Horse(string col)` is a parameterized constructor with one argument namely `col` of type `string`. The statement '`color = col`', is used to assign the value passed in argument `col` at runtime, to the data field `color`. The statement '`Horse h1 = new Horse("Black")`', uses the `new` operator to assign memory to the object `h1` of class `Horse`. Also, the statement '`new Horse("Black")`', invokes the parameterized constructor `Horse(string col)` as soon as the object `h1` gets created. When the constructor is invoked, the value 'Black' gets assigned to the parameter `col`, thereby initializing the data field `color` within the constructor body.

In the first case, that is no-argument constructor, each time a new object of class `Horse` is created, by default 'Brown' will be assigned to the variable `color`. Whereas, in the second case, using parameterized constructor, the value of `color` variable will be set according to the value passed by the user while creating the object.

3.3.4 Static Data Fields

In many cases, it can be easier to have a common data field that is shared by all objects of a class. Normally, the variables that you declare in a class are those that have a separate copy for each object of the class. Such variables are called instance variables. A static or class variable is one for which there is only a single copy that is shared by all objects of a class. For example, the field `color` of class `Horse` can be made shared so that each object will use the same copy of the attribute.

In OOP, shared data fields are declared using the `static` keyword. The C# code given in Code Snippet 6 shows an example of static data fields.

Code Snippet 6:

```
public class Horse
{
    string hname;
    static string color;
  static void showColor()
  {
    Console.WriteLine("Horse color is  "+ color);
  // Console.WriteLine("Horse name is  "+ hname); - error   line1
   }
   static void Main()
  {
     Horse.color = "Black";   // line2
     Horse h1 = new Horse();
     h1.hname = "BlackBeauty";      // line3
    Console.WriteLine("Horse name is "+ h1.hname);  // line4
    Console.WriteLine("Color ="+ Horse.color);  // line5
     Console.WriteLine(Horse.showColor());    // line6
   }
}
```

Output:

```
Horse name is BlackBeauty
Color = Black
Horse color is Black
```

In Code Snippet 6, the class declares two variables namely `hname` and `color`. Here, `hname` is an instance variable, for which each object will have a separate copy, whereas `color` is a static variable for which only a single copy will be created which will be shared by all instances of class `Horse`. To initialize or access a static variable, you need not create an object of the class. A static variable can be directly accessed using the class name as shown in line2 of the example. However, as `hname` is a non-static, or instance variable, it can only be accessed by an object of the class as shown in line3. Similarly, to print the value of a non-static variable, you need to use the object as shown in line4 and for a static variable, use `<class-name>.<variable-name>` as shown in line5.

Similar to static variables, you can also declare a static method and a static class. A class declared as static cannot be instantiated and its members can be accessed directly using the class name.

Concepts

A method declared static, can be accessed directly using the class name without creating object of the class in a similar way as a static variable as shown in line6. However, a static method can only access the static members of a class but not the non-static members. In the code, `showColor()` is a static method, hence it can directly access the static member `color` but it cannot access the non-static member `hname` of the class as shown in line1.

3.3.5 Constant Data Fields

In a class there are certain data fields whose value remains constant throughout the lifetime of an object. Also, each object will have the same value for that data field. For example, each object of the `Horse` class will have same value for the data field `feet`, that is, four. Values of such fields can be fixed before the object of the class is created. Values of such fields will then become constant and will not change with creation of new objects.

Thus, a constant is a variable whose value is fixed during compilation and does not change later. Constant variables are declared using the keyword `const` in C#. In Java, the same effect is achieved using the `final` keyword. Constants need to be initialized when they are declared in C#. However, in Java, initialization of a final variable can be delayed to be done inside the constructor if it is declared at class level or before the end of a method in which it is declared. The C# code given in Code Snippet 7 and Code Snippet 8 shows the creation of a constant.

Code Snippet 7:

```
public class Horse
{
    public const int feet = 4;
    …..
}
```

Code Snippet 8:

```
public class Horse
{
    public const int feet = 4, eyes=2, tail=1;
   ….
}
```

Code Snippet 7 shows a constant data field named `feet` initialized to the value 4. You can also initialize multiple constant data fields of the same type in one line as shown in Code Snippet 8. A constant can be accessed in the same way as any other variable of a class.

For example, one can display value of constant variable `feet` declared in the `Horse` class as shown in the C# code given in Code Snippet 9.

Code Snippet 9:

```
public class Horse
{
   private const int feet=4;

   public void display()
   {
       Console.WriteLine("Number of feet are "+ feet);
    }
    static void Main()
    {
            Horse h1 = new Horse();
            h1.display();
    }
}

Output:
Number of feet are 4
```

In Code Snippet 9, the object `h1` of class `Horse` invokes the `display()` method to print the value of the constant data field `feet`.

3.3.6 Accessor and Mutator

Accessor and Mutator are special methods in OOP that provide an easy yet secure way of accessing a data field of a class. They are also referred to as the getter and setter methods. The aim is to manage values inside the object and avoid misuse by the calling code. Accessor is the method through which the object can retrieve the value of a data field, whereas Mutator is a method through which the object can assign the value to a data field.

Concepts

Figure 3.6 shows Accessor and Mutator methods used to access a data field named `hid` of the class `Horse`.

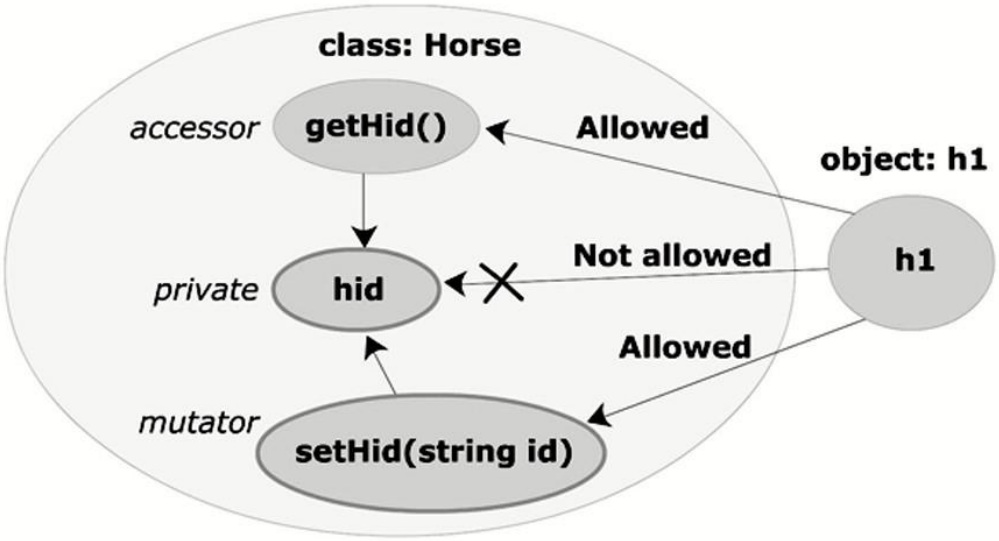


Figure 3.6: Accessor and Mutator

The C# code given in Code Snippet 10 shows the traditional encapsulation using Accessor and Mutator methods.

Code Snippet 10:

```
public class Horse
{
   private int hid = -1;

   public int getHid()       // getter
  {
      return hid;
  }
  public void setHid(int id)     // setter
  {
      hid =id;
  }
```

```
public static void Main()
 {
   Horse h1  = new Horse();
   h1.setHid(10);   // line1
  Console.WriteLine("Horse ID: "+  h1.getHid()); //line 2
  }
}

Output:
Horse ID: 10
```

In Code Snippet 10, the class `Horse` has a private member `hid`. This member is not directly accessible to any object. To set the value of this variable, each object has to make use of the Mutator or setter method `setHid(int id)` as shown in line 1. To print the value of hid, each object has to use the accessor or getter method `getHid()` which will return the value of the variable `hid`. This technique prohibits direct access to the data member `hid` of class `Horse`.

3.3.7 Forward Declaration

Languages such as Java scan the entire file before generating code. This process helps classes that are referenced later in a file to be used at an earlier stage in the code without any conflict. However, languages such as C++ process classes and methods in the sequence that they are encountered. This means that a name should have a partial definition before it is used. Forward declaration of a function is a statement that informs the compiler about the signature (return type, function name, and parameters with their types) of the function without its implementation before the function is used or defined in the code.

In languages such as C, C++, and Pascal, compilers store symbols for variables and functions in a lookup table. They reference this lookup table whenever a function or variable is encountered in the code. As these compilers process the code from top to bottom or sequentially, you need to do forward declaration of a function. This is required because the compiler will find a symbol in the code that it cannot reference in the lookup table. In this case, it will raise an error. Forward declaration is also called a function prototype. It is a prior hint to the compiler that the function definition or implementation has been done later in the code. The C++ code given in Code Snippet 11 shows an example of function prototype.

Code Snippet 11:

```
#include <iostream.h>
// function prototype
void display(string col);  // line1
static void Main ()
{
  string color;
```

Concepts

```
   cout << "Enter color ";
   cin >>color ;
   display(color);  // line 2
} // end of main()
void display (string col)   // line 3
{
  cout<<"Color is "+ col;
}
```

Output:

```
Enter color
black
Color is black
```

Code Snippet 11 shows the use of function `display(string col)` within the body of `main()` in line 2. However, the definition of `display(string col)` comes later as shown in line 3. The code does not issue an error even though the function `display(string col)` is used before it is defined. This is due to the function prototype of `display(string col)` declared prior to its usage as shown in line 1. This is known as forward declaration. The compiler knows in advance that a function with a particular signature is going to be used in the code prior to its definition.

Here, a question may arise as to why the compiler cannot make two passes on each file? - First to index the symbols and second to parse the references. The reason is, when C was invented in 1972, resources such as memory were very scarce. The amount of memory required to store an entire symbolic table for a complex program was not available. The fixed storage was very expensive and slow, so that concepts such as virtual memory and storing of parts of symbolic table on disk would take a long time for compilation. While dealing with storage area such as magnetic tape in which seek times are measured in seconds and throughput in bytes per second (instead of KB or MB), it is quite understandable. C++, which was created as a superset of C, had to use the same mechanism of function prototype.

When Java and C# were invented, the computers had enough memory to store entire symbolic table for a complex project. To avoid the compartmentalization of symbolic table in both Java and C#, the identifiers are automatically recognized from the source code files and directly read from the symbol library. This removes the burden of header files from these languages.

Summary

- A class is a type or a software construct that encloses the data members and functions of an entity into an enclosed structure.
- A visibility modifier is used with a class, data field as well as a method to restrict access to them from outside the class.
- A method defines the behavior of a class and consists of the actions that an object of a class can perform on the data members.
- A constructor is a specialized method that has the same name as the class name and is used to initialize its data members.
- A static or class variable is one for which there is only a single copy and it is shared by all objects of a class.
- A constant is a variable whose value is fixed during compilation and does not change later.
- Forward declaration of a function is a statement that informs the compiler about the signature of the function without its implementation.

Check Your Progress

1. Which of the following option is used with class, data fields as well as methods to restrict access to them from outside the class?

A)	Visibility modifier
B)	Class
C)	Inheritance
D)	Property

2. Which of the following options about a constructor are true?

A)	Constructor is a method that has the same name as the class name
B)	Constructor is used to destroy the data members of a class
C)	Constructor can return a value
D)	Constructor cannot be inherited in child class

3. _____ is a variable whose value is fixed during compilation and does not change later.

A)	Static
B)	Constant
C)	Public
D)	Protected

4. Which of the following option is a variable for which there is only a single copy and it is shared by all objects of a class?

A)	Static
B)	Constant
C)	Final
D)	Local

Check Your Progress

5. Match the following concepts with the corresponding code snippets.

Concept		Code Snippet	

A)	Forward declaration	1)	`class Employee`
			`{`
			`  private int eid;`
			`  public int Eid`
			`  {`
			`  get { return eid;}`
			`  set { eid = value;}`
			`  }`
			`}`

B)	Constructor	2)	`private int id;`

C)	Variable declaration	3)	`public class Hello`
			`{`
			`  Hello()`
			`  {`
			`  Console.WriteLine("Hello World");`
			`  }`
			`}`

D)	Property	4)	`void add (int a, int b);`
			`void main ()`
			`{`
			`  .....`
			`  add(x,y);`
			`}`
			`  void add (int a, int b)`
			`{`
			`  .....`
			`}`

Try It Yourself

1. St. Thomas is a famous institute in the city Chicago. There are many courses taught in the institute such as Information Technology, Finance, Multimedia, and Networking. The students can enroll for any course they desire. Mr. Zen is a faculty of Information Technology in the institute. There are many divisions in the institute such as Front office, Back office, Accounts, Faculty room, and Class room. Identify and list the various classes from the given information.

Session 4

Abstraction and Inheritance

Objectives

At the end of this session, the student will be able to:

- *Define Abstraction*
- *Explain Levels of Abstraction*
- *Define Inheritance*
- *Explain Types of Inheritance*
- *Explain Variants in Inheritance*
- *List the Advantages of Inheritance*

4.1 Introduction

Consider an average person's understanding of a movie screen. According to that person, it is used to focus the light of the projector to display the movie which is playing in the background. The person does not know how the movie is playing; what is the science behind keeping such a large screen, or why the projector is kept far away from the screen. This means that the details of the screen and the mechanism used for playing the movie are hidden or abstracted from the person to reduce the complexity involved in the process. Also, it is possible to group objects into a category based on their characteristics. For example, a movie theater is a type of building. There are many other types of buildings such as schools, offices, and shops. One can arrange these objects according to their type by grouping the common characteristics in one class and describe their special characteristics at the next level. That is you can organize the classes into a hierarchy based on the concept of inheritance.

This session explains the concept of abstraction in which the irrelevant details of an object are purposefully hidden from the user to simplify the usage of the object. Further, this session describes the various levels and types of abstraction. Finally, the session explains the concept of inheritance, its various types, and advantages.

4.2 Abstraction

Abstraction is the deliberate omission of some aspects of an object or process in order to bring more clarity to other aspects or details of that object. It is not necessary to study an object at the most intricate level in order to understand its structure or mechanism. An object can be studied at a higher level of abstraction which simplifies the understanding of its structure. This purposeful suppression of details about an object is also called Information hiding.

For example, when a person uses a computer, he/she does not know how the signals are passing from the keyboard to the various hardware components of the CPU as shown in figure 4.1.

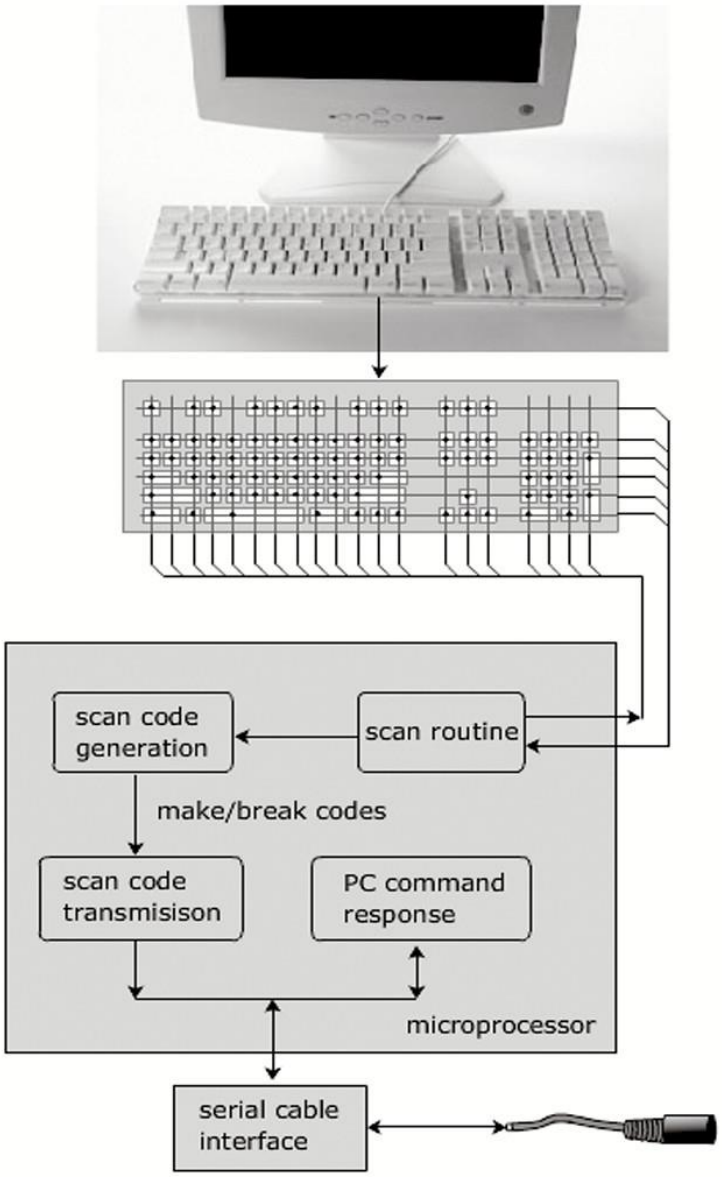


Figure 4.1: Abstraction Through Keyboard

Figure 4.1 shows the abstraction of the complex process of signal transmission by the keyboard.

Similarly, when a person drives a car, he/she simply turns the key in the ignition and the car engine starts. However, the person does not know how this happens. This is abstraction which means to show only the details which the user is concerned with. This is shown in figure 4.2.

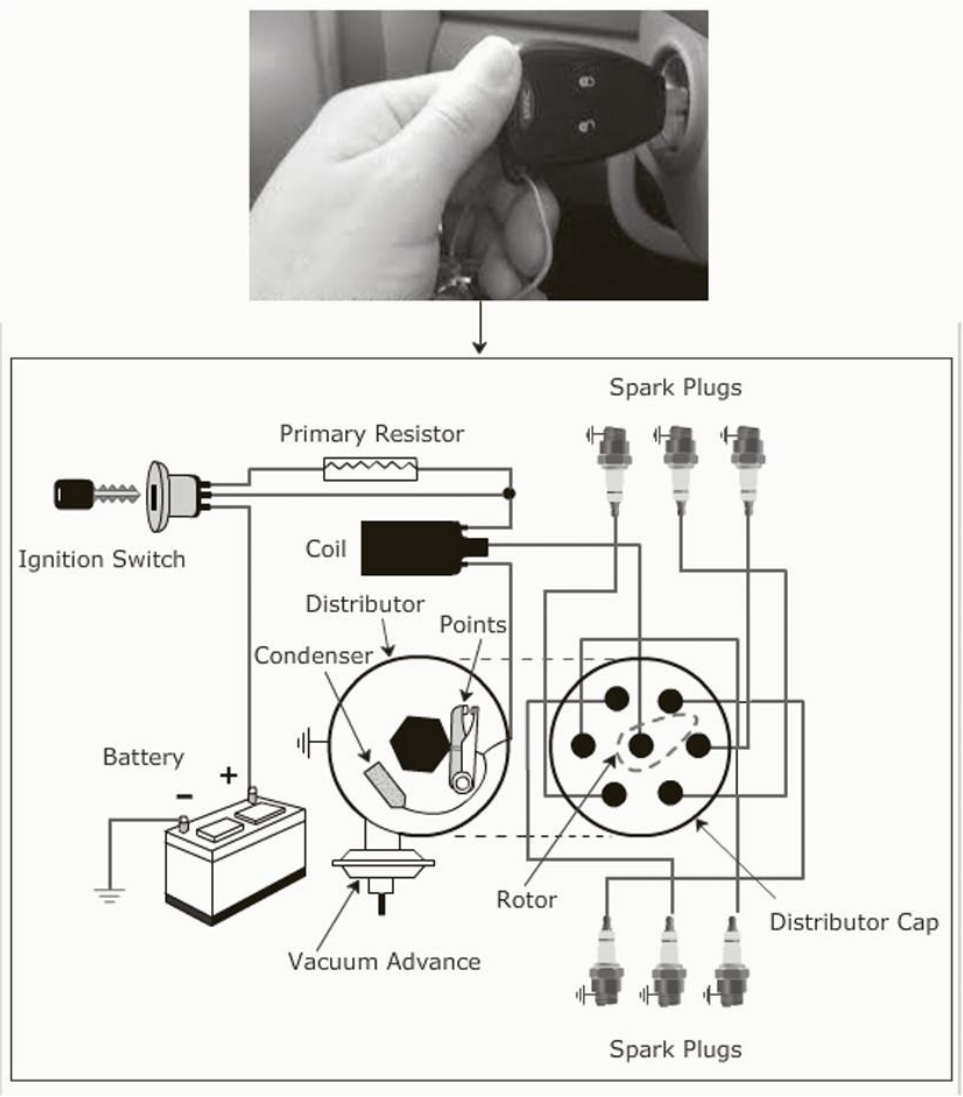


Figure 4.2: Abstraction Through Ignition Lock

Figure 4.2 shows abstraction of the ignition process created by the ignition lock.

Concepts

In Object-oriented programming, abstraction is used to expose only those details of a class that are relevant to the user. The C# code given in Code Snippet 1 shows an example of abstraction.

Code Snippet 1:

```
using System;
public class Horse
{
    static void Main()
    {
            Console.WriteLine("This is class Horse");
    }
}
```

Code Snippet 1 defines a class called `Horse`. Within this class is the `Main()` method that contains a statement '`Console.WriteLine("This is class Horse")`'. This statement is used to print the value '`This is class Horse`' on the console. Here, the inner working of the method `WriteLine()` is not known to the user. This method is defined in the class `Console`. The user has to only pass a parameter to this method and it will be printed on the console. The inner working of the method is hidden from the user or abstracted from the user. Thus, the client using the `Console` class need not be aware of the details of how the method `WriteLine()` has been implemented.

4.2.1 Levels of Abstraction

In a program created in the object-oriented manner, there are several levels of abstraction. These levels are as follows:

- **At the topmost level, a program is viewed as a collection or community of agents or objects interacting with each other to complete a task**.

  For example, a community of developers, who must interact with each other to develop a software is shown in figure 4.3.

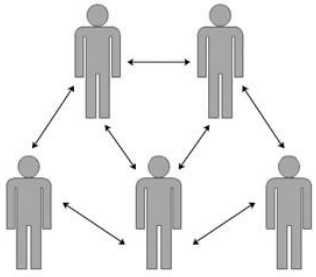


Figure 4.3: Level 1 Abstraction

Each object in this community provides a service that is used by other members to complete their task.

- **In the next level of abstraction objects working together or having common characteristics are grouped into a unit. For example, a package in Java or an assembly in .NET.**

The unit allows certain objects to be exposed to the objects outside the unit, while other features remain hidden inside the unit. This is shown in figure 4.4.

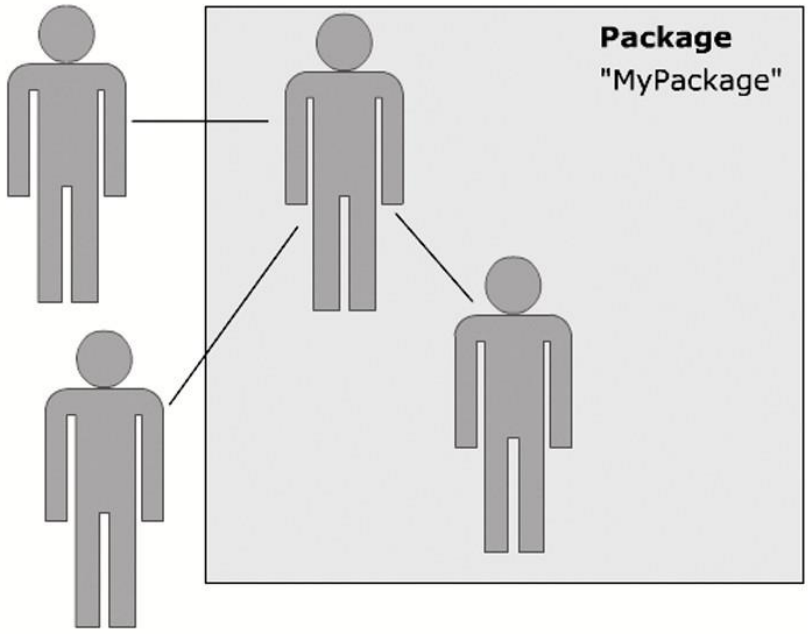


Figure 4.4: Level 2 Abstraction

However, all object-oriented programming languages may not support this level of abstraction.

Concepts

- **The next level of abstraction deals with interactions between two objects.**

  An object often provides services to other objects. Such an object is called a server and the object receiving the service is called the client. This is shown in figure 4.5.

Note: Here, the term server is not used in the technical meaning of a Web server. Instead, here the term `server` simply means something that is providing a service to others.

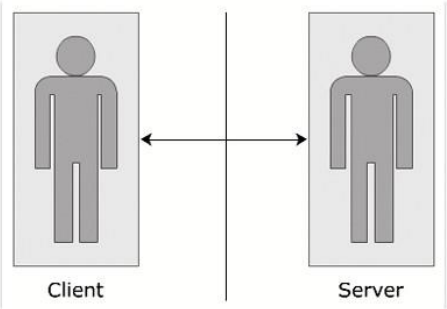


Figure 4.5: Level 3 Abstraction

The two layers of abstraction created here refer to the two views of a relationship, that is, the client side view and the server side view. The client's view can only see the signature of the methods exposed by the server. The server's view refers to the concrete implementation of the methods or services that it has exposed to the client.

- **The last level of abstraction deals with a single task or method in isolation**.

  This level is concerned with the exact sequence of operations to be done to perform one task. For example, details of operations to be done to check if a number is even or odd. This is shown in the C# code given in Code Snippet 2.

Code Snippet 2:

```
public class CheckNum
{
      public void check(int num)
      {
           if(num % 2 == 0)
              Console.WriteLine("Even");
           else
              Console.WriteLine("Odd");
       }
}
```

Code Snippet 2 shows a class called `CheckNum` with a method named `check()`. The code within the `check()` method verifies whether a given number is even or odd and accordingly gives the output.

It is difficult to find the right level of abstraction in the early stages of software development. However, the higher the level of abstraction, the more object-oriented a program is. Each level of abstraction can be useful at some point during the development process.

➢ **Other forms of Abstraction**

The two most important forms of abstraction used in object-oriented programming are division into parts and division into specializations. These forms of abstraction are as follows:

- Division into parts is called has-a abstraction. For example, Car has-a Steering and Horse has-a tail. The has-a abstraction shows container-content or part-of relation between the objects. That is steering is contained inside or is a part-of the car. Figure 4.6 shows an example of 'has-a' abstraction.

Figure 4.6: Has-a Abstraction

- Division into specializations is called is-a abstraction. For example, Horse is-a Herbivore and Car is-a Vehicle. The is-a abstraction shows type-of or kind-of relation between the objects. That is house is a kind of building. Figure 4.7 shows an example of 'is-a' abstraction.

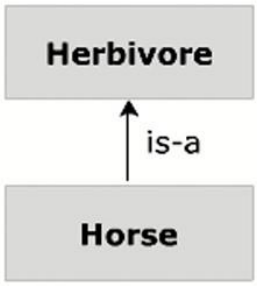


Figure 4.7: Is-a Abstraction

Concepts

4.3 Inheritance

Inheritance is a technique in object-oriented programming in which a user can extend the functionality of a class by creating a new class. The newly created class is called the sub-class or derived class and the class from which it extends is called the super-class or base class. Thus, the two classes share a parent-child relationship. The child class inherits the data fields and methods of the parent class. However, the child class can also create its own data fields and methods. Inheritance helps a user to bundle the common characteristics of a set of classes into a separate class and the specific characteristics are defined in the individual classes. The parent-child relation is also called 'is-a' relationship. Figure 4.8 shows a real world example of inheritance.

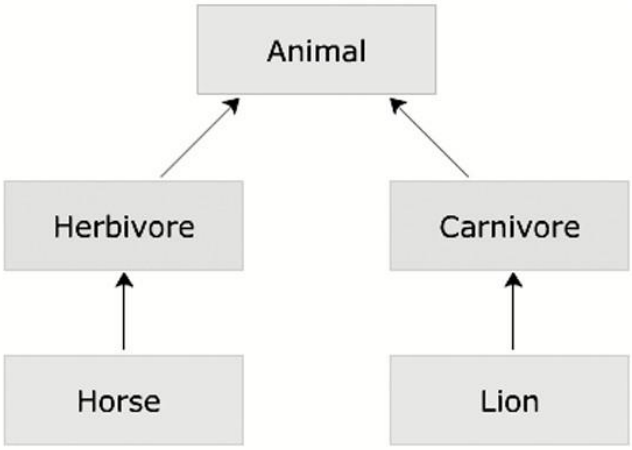


Figure 4.8: Inheritance

Figure 4.8 shows that `Horse` is a type of `Herbivore` and `Lion` is a type of `Carnivore`. Both `Herbivore` and `Carnivore` are of type `Animal`. The common characteristics of all Horses and Lions are bundled in classes `Herbivore` and `Carnivore` respectively. Similarly, common characteristics of all `Herbivores` and `Carnivores` are bundled in the class `Animal`. The specific characteristics of `Horse` and `Lion` will be defined in the respective classes.

A class can inherit features from a super class at several levels. This gives rise to different types of inheritance such as Single, Multiple, Multilevel, Hierarchical, and Hybrid inheritance. Each type has its own characteristics and implementations in various programming languages.

Inheritance does not imply that all features of a parent class will be inherited by the child class. It depends on the access modifiers attached to each data member or method of the parent class. The members declared `private` in the parent class cannot be inherited by the child class.

4.3.1 Single Inheritance

When a class derives from only one base class, it is called single inheritance. This is shown in figure 4.9.

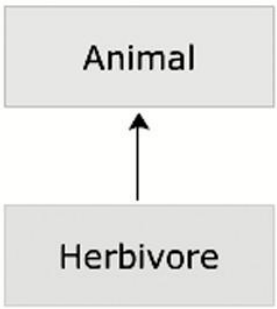


Figure 4.9: Single Inheritance

In figure 4.9, class `Herbivore` has only one parent class namely class `Animal`. Therefore, it is an example of single inheritance.

4.3.2 Multilevel Inheritance

When a class derives from another derived class, it is called multilevel inheritance. This is shown in figure 4.10.

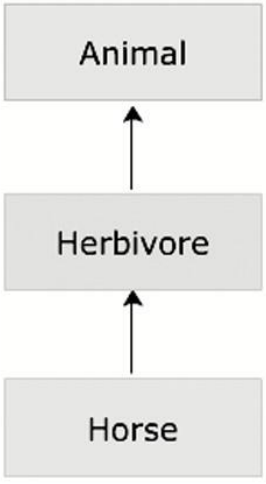


Figure 4.10: Multilevel Inheritance

Concepts

In figure 4.10, class `Horse` is a sub-class of class `Herbivore` and class `Herbivore` is a sub-class of class `Animal`. This is called multilevel inheritance. Here, features of class `Animal` will be inherited by class `Herbivore` and features of class `Herbivore` will be inherited by class `Horse`. Features of class `Animal` will also be inherited by class `Horse` through class `Herbivore`. This is called multilevel inheritance.

4.3.3 Multiple Inheritance

When a class derives from more than one base class it is called multiple inheritance. This is shown in figure 4.11.

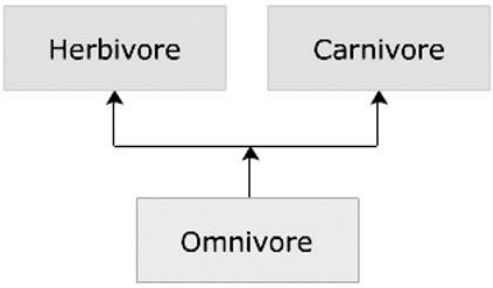


Figure 4.11: Multiple Inheritance

In figure 4.11, class `Omnivore` inherits class `Herbivore` as well as class `Carnivore`. Hence, it will directly inherit features of both parent classes. This is called Multiple Inheritance. Languages such as C# and Java do not support multiple inheritance.

4.3.4 Hierarchical Inheritance

When more than one child class derives from one base class it is called hierarchical inheritance. This is shown in figure 4.12.

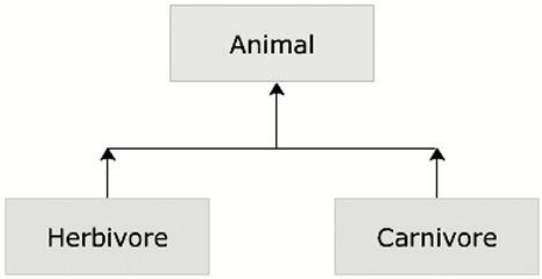


Figure 4.12: Hierarchical Inheritance

In figure 4.12, classes `Herbivore` and `Carnivore` both extend from class `Animal`. This is called hierarchical inheritance.

4.3.5 Hybrid Inheritance

When a class derivation involves more than one form of inheritance, it is called hybrid inheritance. This is shown in figure 4.13.

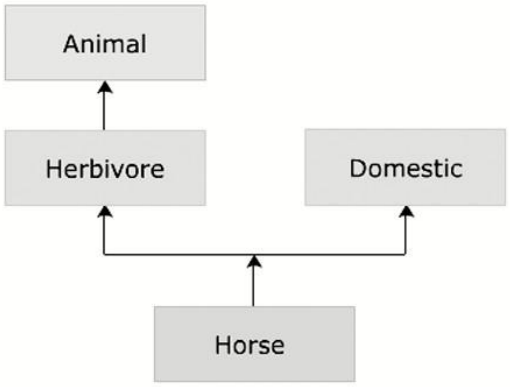


Figure 4.13: Hybrid Inheritance

In figure 4.13, class `Horse` inherits from classes `Herbivore` and `Domestic`, which is a multiple inheritance. Class `Herbivore` in turn inherits from the class `Animal` so that features of class `Animal` are passed on to class `Horse` through class `Herbivore`. This is multilevel inheritance. Thus, this is a combination of two different types of inheritance and therefore, can be termed Hybrid inheritance.

All object-oriented programming languages have their own methodology and syntax for implementing the concept of inheritance. The C# code given in Code Snippet 3 shows an example of single inheritance.

Code Snippet 3:

```
public class Herbivore
{
  protected string name;      // line1
  private string food = "grass";   // line2
  pubic void display()
  {
    Console.WriteLine("It eats "+ food);
  }
}
```

Concepts

```
// child class inherits from parent class
class Horse : Herbivore
{
   static void Main()
  {
     Horse h1 = new Horse();
     h1.name = "Black Beauty";    // line3

     Console.WriteLine("Horse name is " + h1.name); // line4

     h1.display();  // line5
   }
}
```

Output:

```
Horse name is Black Beauty
It eats grass
```

Code Snippet 3 shows a class `Herbivore` having two variables `name` and `food`. Variable '`name`' is declared `protected` as shown in line1 and '`food`' declared `private` as shown in line2. Class `Herbivore` also has a method `display()`that prints the value of variable `food`. Another class named `Horse` inherits from `Herbivore`. Now, according to the rule, a child class has access to the public and protected members of its parent but not the private members. Therefore, object of class `Horse` can directly access the variable `name` as shown in line3 and line4 but it cannot directly access variable `food` since it is declared private in the parent class. To print the value of variable `food`, the object has to make use of the public method `display()` which it has inherited from class `Herbivore` as shown in line5.

4.4 Variants in Inheritance

Inheritance can have several implementations such as Anonymous class, constructors with inheritance, and Inner classes.

4.4.1 Anonymous Class

In some cases a programmer may require to create a class that should have only one instance. Such an object is called a singleton. Programming languages such as Java provide a technique for creating an object of a class without having to give a name to the class whose object is being created. Such a class is called an anonymous class as it does not have an identity. To create an anonymous class, the user has to follow some rules. That is, an anonymous class can have only one instance; the class must inherit from some parent class and does not require a constructor for initialization.

4.4.2 Constructors and Inheritance

A constructor is a method with the same name as the class name and is invoked automatically when a new instance of a class is created. It ensures proper initialization of the newly created object. However, due to inheritance, the situation may become complicated as both parent and child classes have their own constructors and initialization code. This means that constructors of both classes must be executed when the object of child class is created. In C#, Java, and other such object-oriented languages, parent class constructor is invoked automatically before child class constructor or the child class constructor can explicitly invoke the parent class constructor. The C# code given in Code Snippet 4 shows an example of constructor execution with inheritance.

Code Snippet 4:

```
class Herbivore
{
    // parent class constructor
    public Herbivore
    {
        Console.WriteLine("Class Herbivore");
    }
}
// Child inherits Parent
class Horse : Herbivore
{
    // child class constructor
     public Horse
    {
        Console.WriteLine("Class Horse");
     }
    static void Main()
    {
        Horse h1 = new Horse();   // line1
    }
}
```

Output:

```
Class Herbivore
Class Horse
```

Concepts

In Code Snippet 4, an object of `Horse` class is created using the new operator as shown in line1. Once the object is created, first the `Herbivore` class constructor is invoked and then the `Horse` class constructor. Therefore, in the output, the statement '`Class Herbivore`' is printed first and then the statement '`Class Horse`' is printed.

4.4.3 Inner Classes

Languages such as C++, Java, and C# allow the programmer to create definition of one class inside another class. Such a class is known as inner class or nested class as shown in figure 4.14.

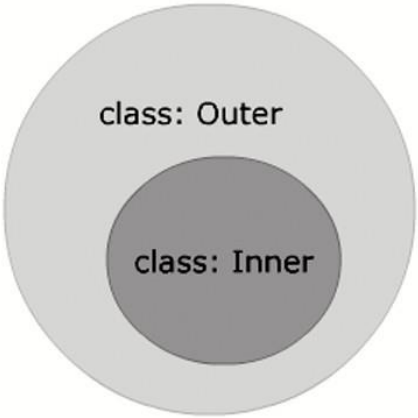


Figure 4.14: Inner or Nested Class

The C# code given in Code Snippet 5 shows an example of nested class.

Code Snippet 5:

```
class Outer    // defining the outer class
{
    public static int x = 10;
    public int y = 20;
  public class Inner   // defining the nested class
 {
   public static void display()
  {
      // accessing outer class static variable
      directly with its name inside the inner class
     Console.WriteLine(Outer.x); // line2
```

```
// accessing outer class variable inside inner class
   using its object
Outer oc = new Outer();  // line3
Console.WriteLine(oc.y);  // line4
   }
 } // inner class ends
} // outer class ends
class Sample   // external class
{
    public static void Main()
    {
        Console.WriteLine("Sample");
 // invoking nested class method from an external class
        Outer.Inner.display();     // line1
    }
}

Output:
Sample
10
20
```

Code snippet 5 shows three classes namely, `Sample`, `Outer`, and `Inner`. Class `Inner` is defined within the class `Outer`. This means that `Inner` is a 'nested' `class`. Class `Sample` is an independent external class. There are several conditions that apply on a class that is defined inside another class. These conditions defer with various languages. In the example, the nested class has been created using C#. The conditions that apply to the nested class in C# are as follows:

- Inner class cannot be accessed directly from an external class. It has to be declared `public` and used with its fully qualified name in the external class. This means that class `Sample` can use the class `Inner` with its fully qualified name as shown in line1 – `Outer.Inner.display();`
- A static variable declared in the outer class can be directly accessed by the inner class using the outer class name as shown in line2 – `Console.WriteLine(Outer.x);`
- It is not mandatory to declare an outer class variable as `static` to use it in the inner class. Instance variables of outer class can also be accessed inside the inner class by creating an object of the outer class inside the inner class as shown in line3 and line4 –

```
Outer oc = new Outer();

Console.WriteLine(oc.y);
```

Concepts

- Nested class can also inherit the outer class. That means you can write a C# code as shown in Code Snippet 6.

Code Snippet 6:

```
class Outer
{
   private int x = 20;
 public class Inner : Outer // inner class inheriting outer class
 {
    public static void display()
    {
        Outer oc = new Outer();
        System.Console.WriteLine(oc.x);
    }
 }
}
```

In Code Snippet 6, nested class `Inner` is inheriting the enclosing class `Outer`.

- If an external class inherits the outer class, the constructors are executed in the usual manner. That means, the base class constructor is executed before the derived class constructor. This is shown in the C# code given in Code Snippet 7.

Code Snippet 7:

```
class Outer
{
    public Outer
    {
        Console.WriteLine("Outer Class");
    }
    public class Inner
    {
        public Inner()
        {
            Console.WriteLine("Inner Class");
        }
    }
}
```

```
class Sample : Outer   // line1
{
    public static void Main()
    {
        Sample s1 = new Sample();     // line2
        Console.WriteLine("Sample");
    }
}

Output:
Outer class
Sample
```

In Code Snippet 7, class `Sample` inherits class `Outer` as shown in line1. Therefore, when an object of `Sample` class is created in line2, first the constructor of class Outer is invoked as it is the parent class and then the constructor of class `Sample` is invoked.

Note: The class `Inner's` constructor will not be invoked since the class has not been referenced.

4.5 Advantages of Inheritance

Some advantages of Inheritance are as follows:

- **Reusability** – When a class inherits the data members and methods of another class, it can reuse them without having to write the functionality again. This is known as code reuse. This saves the time during development and one can focus on the more specific areas of the code. For example, consider a class named `Calc` that has a public method named `add()`. Now, when a class named `ScientificCalc` inherits class `Calc`, it has direct access to the `add()` method for doing addition of two numbers. It need not write the functionality of adding two numbers again in its own class as it can achieve that using the method `add()` inherited from its parent class.
- **Consistency** – When a class inherits another class any changes made in the behavior of the parent class will automatically get reflected in the child class. This ensures that the inherited behavior will remain same or consistent for all child classes that inherit the parent class.
- **Modularity** – Inheritance provides a means to make code modular and reusable. It divides the functionality amongst various components which can be used later in other programs also. The idea is to enable development of new applications without having to write the same code again. The reusable components can be bundled into libraries and made readily available for reuse.
- **Security** – A programmer using a software library only has access to the method signatures but not to the method implementation. That means one only needs to know the interface or nature of the component but not the detailed information about its inner working mechanism. This gives some amount of security to the code and prevents misuse or corruption by unauthorized external components.

Summary

- Abstraction is the deliberate omission of some aspects of an object or process in order to bring more clarity to other aspects or details of that object.
- Division into parts is called has-a abstraction as it shows container-content or part-of relation between the objects.
- Division into specialization is called is-a abstraction as it shows type-of or kind-of relation between the objects.
- Inheritance is a technique in object-oriented programming in which a user can extend the functionality of a class by creating a new class.
- Single Inheritance is a type of inheritance in which a class derives from only one base class.
- A class derivation that involves more than one form of inheritance is called hybrid inheritance.
- A class defined inside another class is known as an inner class or nested class.

Check Your Progress

1. Match the following Inheritance type with the corresponding description.

Inheritance type		Description	
A)	Multiple Inheritance	**1)**	An inheritance that involves more than one form of inheritance.
B)	Multilevel Inheritance	**2)**	A class inherits from more than one parent classes.
C)	Single Inheritance	**3)**	A class inherits from another derived class.
D)	Hybrid Inheritance	**4)**	A class inherits from only one parent class.

2. Which of the following option about inheritance is true?

A)	Inheritance allows code reuse
B)	Inheritance makes the code insecure
C)	Inheritance helps to maintain consistency in code
D)	Inheritance makes the code modular

3. A class that does not have an identity is _____.

A)	Parent class
B)	Anonymous class
C)	Sub-class
D)	Abstract class

Concepts

Check Your Progress

4. Consider the following code:

```
public class Parent
{
  pubic void display()
  {
    Console.WriteLine("I am parent");
  }
}
class Child : Parent
{
   static void main()
   {
     Child c1 = new Child();
     c1.display();
   }
}
```

What will be the output of the code?

A)	The code will issue a compile time error
B)	The code will give no output
C)	The code will print 'I am parent' on the console
D)	The code will issue a runtime error

5. Match the following objects with its corresponding type using 'is-a' abstraction.

Object		Type	
A)	Car	**1)**	Building
B)	Parrot	**2)**	Vehicle
C)	School	**3)**	Furniture
D)	Table	**4)**	Bird

Try It Yourself

1. Rob is the manager of the IT department in a bank in New York. The management of the bank has decided to automate the various operations and bank transactions. Rob has been assigned the task to create the system design of the bank. Rob studies the system and comes to the conclusion that there are two kinds of employees in the bank; Full time and Part time. There are departments such as Accounts, HR, and Loan. The customers are allowed to open an account with the bank such as Savings account and Current account. He will require creating association between all these entities of the bank for the system to be automated. Identify the various classes that can be created from the scenario and draw the inheritance hierarchy to depict association between the classes.

2. Consider the following list of objects. Classify them into is-a and has-a abstraction models.

   King, Engine, Cat, Dog, Vehicle, Table, Chair, Animal, Car, Bus, Furniture, Person, Tail, Steering.

Session 5

Multiple Inheritance and Interfaces

Objectives

At the end of this session, the student will be able to:

- *Describe Multiple Inheritance*
- *List the problems associated with Multiple Inheritance*
- *Describe Interface*
- *Explain Multiple Inheritance using Interfaces*
- *Explain constructor execution in Multiple Inheritance*

5.1 Introduction

The essence behind the concept of inheritance is the 'is-a' relationship. 'Is-a' relationship can be viewed in another way as a classification. For example, a car can be categorized as a Vehicle, and therefore, the class Car can be created by inheriting it from the class Vehicle. However, in the real world, an object may not always fit a single parent hierarchy. This is because a real world object can be classified in multiple and mutually non-overlapping ways. For example, a potter can be a male, an artist, a parent, and an American. Each of these types reveal something about the potter and yet none of them is a correct subset of any of the others. If an inheritance hierarchy is created out of these categories, it may lead to incorrect classifications. For example, it may end up asserting that all males are Americans or all artists are Males, or some other incorrect classification.

This session explains the concept of multiple inheritance which emphasizes the correct way of expressing the relationships between classes. Further it explains how a class can be created by combination of other classes. Finally, the session describes the use of interfaces to achieve multiple inheritance in programming languages that do not have support for multiple inheritance.

5.2 Multiple Inheritance

Multiple inheritance is a technique in which one class inherits its properties from more than one classes. For example, an omnivore is an herbivore which is a plant eater as well as a carnivore which is meat eater. Each category contributes some of its features to the inheriting entity. Hen, pig, cat, and human are some examples of omnivorous animals.

Figure 5.1 shows hen, pig, cat, and human as omnivorous animals.

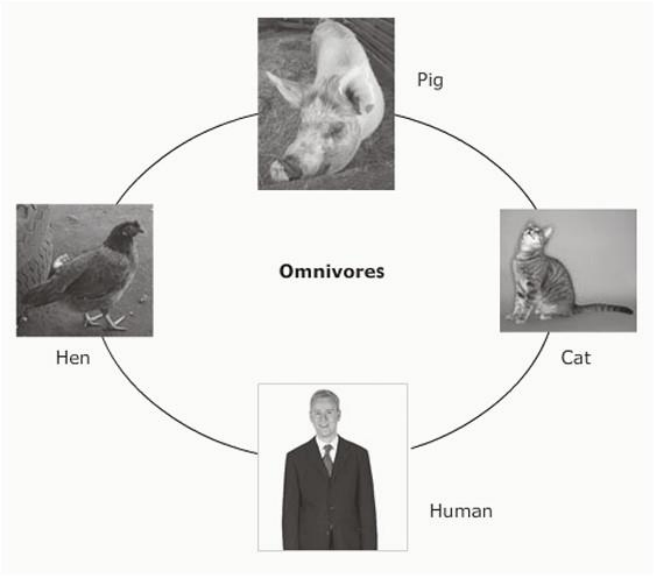


Figure 5.1: Omnivorous Animals

Figure 5.2 shows an example of multiple inheritance.

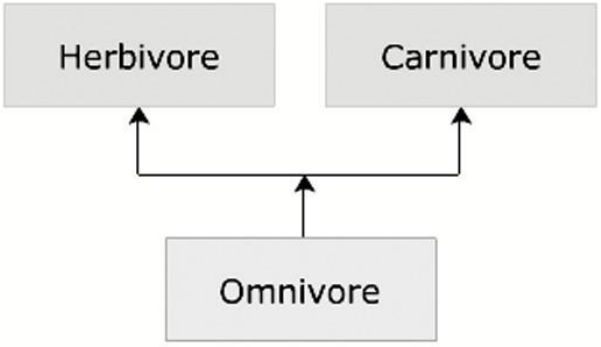


Figure 5.2: Multiple Inheritance

Figure 5.2 shows the class `Omnivore` inheriting from two classes namely `Herbivore` and `Carnivore`. The class `Omnivore` will derive properties of both classes and will also have its own specific features.

OOP languages such as C# and Java do not support multiple inheritance. However, C++ does allow multiple inheritance.

Concepts

The C++ code given in Code Snippet 1 shows an example of multiple inheritance.

Code Snippet 1:

```
#include <iostream.h>
class Herbivore
{
// constructor of class Herbivore
public:
   Herbivore(string pType)
   {
      plantType = pType;
   }
protected:
    string plantType;
};
class Carnivore
{
// constructor of class Carnivore
 public:
   Carnivore(string aType)
   {
      animalType = aType;
   }
  protected:
    string animalType;
 };
// class Omnivore inherits Herbivore and Carnivore
class Omnivore : public Herbivore, public Carnivore
{
public:
   string oName;
  // constructor of class Omnivore
  Omnivore(string name , string pType, string aType)
      : Herbivore(pType), Carnivore(aType)          // line1
 {
       oName = name;  // line2
 }
   void display()
   {
```

```
      //printing the values of the variables
       cout << "Omnivore name is "+ oName << endl;
       cout << "Plant it eats is " + plantType << endl;
       cout << "Animal it eats is "+ animalType<< endl;
    }
};
static void Main()
{
    Omnivore o1('Brian', 'Herbs', 'Chicken');
    o1.display();     // line3
}
```

Output:

```
Omnivore name is Brian
Plant it eats is Herbs
Animal it eats is Chicken
```

In Code Snippet 1, there are three classes namely, `Herbivore`, `Carnivore`, and `Omnivore`. Class `Herbivore` has a protected variable named `plantType` and class `Carnivore` has a protected variable named `animal`. Both classes are using constructors to initialize the values of the respective variables. Class `Omnivore` is inheriting both the classes `Herbivore` and `Carnivore`. It has its own public variable called `oName`. The constructor of class `Omnivore` takes three parameters namely, `name`, `pType`, and `aType`. The `pType` and `aType` variables are passed to the constructors of classes `Herbivore` and `Carnivore` respectively as shown in line1. The name variable is used to initialize the `oName` variable as shown in line2. In the `main()` method, an object of class `Omnivore` is created to invoke the method `display()`. The `display()` method displays the value of variable `oName` as well as `plantType` and `animalType` which have been inherited by class `Omnivore`.

The class diagram of Code Snippet 1 is shown in figure 5.3.

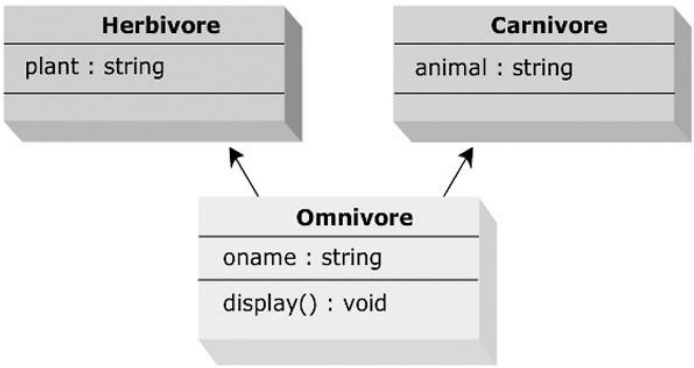


Figure 5.3: Class Diagram of Multiple Inheritance

Concepts

Figure 5.3 shows the class diagram of the inheritance hierarchy created by the three classes namely, `Herbivore`, `Carnivore`, and `Omnivore`.

5.2.1 Problems with Multiple Inheritance

Some problems arising from multiple inheritance are as follows:

- **The Diamond Problem** - In multiple inheritance, it may happen that a class inherits from two parent classes, each of which in turn inherit from a common super class. Suppose class D inherits from classes B and C. Classes B and C inherit from a common class A. In this case, both B and C have inherited members of class A. Now, when D inherits B and C, it will have two copies of members of class A. This gives rise to an ambiguity that from which class will it inherit members of class A, B, or C? This is known as the Diamond Problem.

  For example, if one level is increased in the inheritance hierarchy shown in figure 5.2, to include the `Animal` class as a super class of classes `Herbivore` and `Carnivore`, the resulting hierarchy would be as shown in figure 5.4.

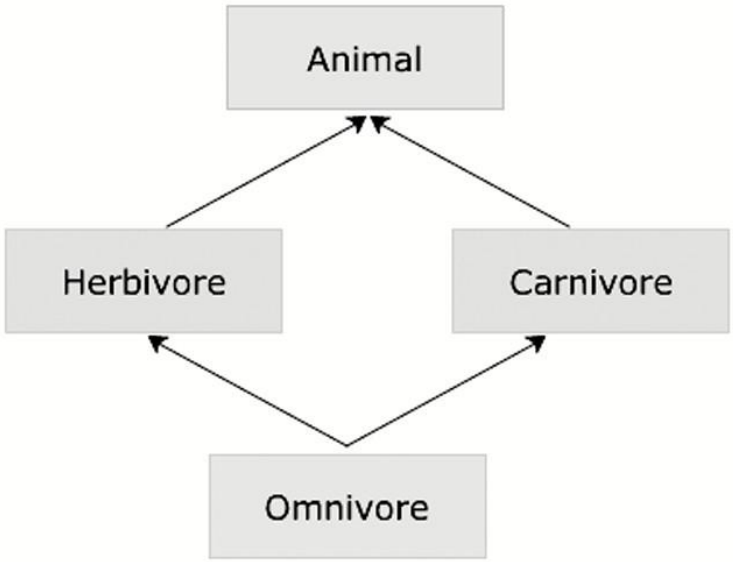


Figure 5.4: The Diamond Problem

Figure 5.4 shows the diamond problem resulting from a common parent class `Animal` from which classes `Herbivore` and `Carnivore` are derived. It is termed as 'The Diamond Problem' because of the resulting shape which looks like a diamond. This shows that class `Omnivore` will contain two copies of members of class `Animal`; one from class `Herbivore` and another from class `Carnivore`.

The various languages that support multiple inheritance, deal with this problem in different ways. C++ resolves this problem by marking the common parent class as `virtual` when the intermediate child classes inherit it. In figure 5.4, classes `Herbivore` and `Carnivore` will have to inherit the super class `Animal` marked `virtual` as shown in the C++ code given in Code Snippet 2.

Code Snippet 2:

```
#include <iostream>
class Animal
{
   protected:
       int animalId;
}
// inheriting class Animal as virtual
class Herbivore : virtual public Animal
{
   protected:
    string plantType;
};
// inheriting class Animal as virtual
class Carnivore : virtual public Animal
{
  protected:
    string animalType;
 };
// class Omnivore inherits Herbivore and Carnivore
class Omnivore : public Herbivore, public Carnivore
{
public:
   string oName;
  void setValues(int id, string name, string pType, string aType)
   {
          animalId = id;
          oName = name;
          plantType = pType;
          animalType = aType;
   }
   void display()
   {
      //printing the values of the variables
```

```
        cout << "Omnivore ID is  "+ animalId << endl ;
        cout << "Omnivore name is  "+ oName << endl ;
        cout << "Plant it eats is " + plantType << endl;
        cout << "Animal it eats is "+ animalType<< endl;
    }
};
static void Main()
{
    Omnivore o1;
    o1.setValues(100,"Brian","Herbs","Chicken");
    o1.display();
}

Output:
Omnivore ID is 100
Omnivore name is Brian
Plant it eats is Herbs
Animal it eats is Chicken
```

In Code Snippet 2, classes `Herbivore` and `Carnivore` inherit class `Animal` marked as `virtual`. Here, the C++ compiler will take extra care by creating only one copy of class `Animal` to ensure that the classes further down in the hierarchy do not get duplicate copies of class `Animal`. Therefore, class `Omnivore` will have only one copy of variable `animalId` of the `Animal` class. This resolves the Diamond Problem.

- **Ambiguity of name** - In multiple inheritance, a situation may arise where the base classes have a method with the same name and signature. In such a case, the derived class will inherit both the methods but the compiler will not know which one to invoke when the child class references it. For example, suppose in both classes `Herbivore` and `Carnivore` in figure 5.3, the user adds a method named `display()` as shown in figure 5.5.

Figure 5.5 shows the name ambiguity problem in multiple inheritance.

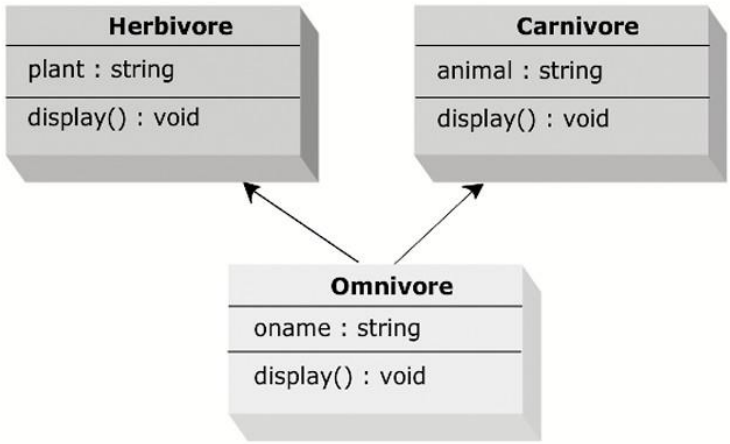


Figure 5.5: Name Ambiguity Problem in Multiple Inheritance

Figure 5.5 shows that both classes `Herbivore` and `Carnivore` have a method named `display()` with the same signature. In this case, the compiler will give error when an object of class `Omnivore` tries to invoke the `display()` method as shown in the C# code given in Code Snippet 3.

Code Snippet 3:

```
Omnivore o1 = new Omnivore();
o1.display();   // error due to ambiguity
```

C++ uses two ways to solve this problem. They are as follows:

1. Use a different name for the methods in the parent classes. That is, one method can be named `display()` and other can be named `show()`.

2. Use scope resolution operator '`::`' with the base class to identify which `display()` method is invoked. For example:

```
Omnivore *o1 = new Omnivore();

o1 -> Herbivore :: display();

o1 -> Carnivore :: display();
```

Concepts

5.3 Multiple Inheritance with Interfaces

Programming languages such as Java and C# do not support multiple inheritance of classes, but both allow implementation of multiple inheritance through interfaces. An interface is a type similar to a class, but unlike a class, an interface does not supply code to its child class. An interface only declares the signature of the methods it wishes to expose. The implementation of the methods has to be done by the class that implements the interface. Thus, methods in an interface do not have a body. Such methods are called abstract methods.

Interface declaration is similar to a class. The syntax of an interface is shown here.

Syntax:

```
<visibility-modifier> interface <interface-name>
{
      // abstract method declarations
}
```

where,

`visibility-modifier`: States the visibility of the interface.

`interface-name`: The name of the interface being declared.

`method declarations`: The member methods of the interface.

The structure of an interface contains a `visibility-modifier` followed by the keyword '`interface`' and `interface-name`. The two curly braces '`{ }`' mark the start and end of the limit of the interface. All abstract methods are declared within the curly braces.

Though the structure of a class and an interface look similar, there are certain features that differentiate them. They are shown in table 5.1.

Class	Interface
A class can be instantiated	An interface cannot be instantiated
A class can have concreted methods, that is, methods with a body	An interface cannot have concrete methods
A child class may not override all methods of a parent class	A class implementing an Interface must implement all methods of an interface
Multiple classes cannot be inherited by a child class to simulate multiple inheritance	Multiple interfaces can be implemented by a class to simulate multiple inheritance
An 'is-a' relationship is required to implement inheritance between classes	An 'is-a' relationship is not mandatory for implementing an interface

Class	Interface
Methods of an interface are implicitly public	Methods of a class can have any access modifier

Table 5.1: Difference Between Class and Interface

The OOP languages such as Java and C# have their own way of creating an interface. The C# code given in Code Snippet 4 shows the creation of an interface.

Code Snippet 4:

```
interface IHerbivore
{
    void display();
}
```

Code Snippet 4 shows an interface named `IHerbivore` declared in C# along with its method `display()`. The method `display()` is an abstract method, that is, a method without a body. In contrast to the `display()` method declared in the class `Herbivore` in figure 5.5, the `display()` method of interface `IHerbivore` is abstract, so that its implementation has to be done by the child class that implements interface `IHerbivore`.

Note: An abstract method is one that only contains the method signature and no method body.

Similarly, there can be another interface named `ICarnivore` with the same method named `display()`. In figure 5.5, the class `Omnivore` is shown to inherit two classes namely `Herbivore` and `Carnivore`, both of which have their own definition for method `display()`. This leads the problem of ambiguity in C++. This problem is resolved by the use of interface as interface methods are abstract. So that even if a class implements two interfaces having methods with the same signature, only one copy will be used by the child class as shown in the C# code given in Code Snippet 5.

Code Snippet 5:

```
interface IHerbivore
{
    void display();
}
interface ICarnivore
{
    void display();
}
// class Omnivore implements interfaces IHerbivore and ICarnivore
```

Concepts

```
class Omnivore : IHerbivore, ICarnivore
{
    void display()  // line1
    {
     Console.WriteLine("I am an Omnivore");
    }
}
```

Code Snippet 5 shows two interfaces namely `IHerbivore` and `ICarnivore`; both having method `display()`. Class `Omnivore` implements the two interfaces and derives the method `display()` from them. However, implementation of the method is not provided by the interfaces but has to be done by the class `Omnivore` as shown in line1. Also, it needs to provide implementation of only one `display()` method.

However, if it is required that the two interfaces should have two different `display()` method implementations, then the developer must change the signature of `display()` method in any one of the interfaces. In that case the child class will have to implement both the `display()` methods separately as shown in Code Snippet 6.

Code Snippet 6:

```
interface IHerbivore
{
    void display();
}
interface ICarnivore
{
    void display(string name);
}
// class Omnivore implements interfaces IHerbivore and ICarnivore
class Omnivore : IHerbivore, ICarnivore
{
    void display()  // line1
    {
     Console.WriteLine("I am an Omnivore");
    }
     void display(string name)  // line2
    {
     Console.WriteLine("Omnivore.name is "+ name);
    }
}
```

Code Snippet 6 shows two interfaces namely `IHerbivore` and `ICarnivore`; both having method `display()` but with different signatures. Class `Omnivore` implements the two interfaces and derives two `display()` methods from them. Also, the class `Omnivore` must give separate implementations of both the `display()` methods as shown in line1 and line2.

The language Java uses the keyword '`implements`' to show that a class implements interfaces as shown in Code Snippet 7.

Code Snippet 7:

```
class Omnivore implements IHerbivore, ICarnivore
{
   // ….
}
```

An interface can inherit other interfaces. In such a case, the class implementing the derived interface will have to implement both parent, as well as child interface methods as shown in the C# code given in Code Snippet 8.

Code Snippet 8:

```
interface IAnimal
{
     void displayID(int animalId);
}
interface IHerbivore : IAnimal
{
      void displayPlant(string plantType);
}
interface ICarnivore : IAnimal
{
      void displayAnimal(string animalType);
}
// class Omnivore implementing interfaces IHerbivore and ICarnivore
class Omnivore : IHerbivore, ICarnivore
{
    void displayID(int animalId)  // line1
    {
     Console.WriteLine("Omnivore id is "+ animalId);
    }
```

Concepts

```
void displayPlant(string pType)  // line2
    {
     Console.WriteLine("Plant it eats is "+ plantType);
    }
    void displayAnimal(string aType)  // line3
    {
     Console.WriteLine("Animal it eats is "+ animalType);
    }
}
```

Code Snippet 8 shows that interfaces `IHerbivore` and `ICarnivore` inherit the interface `IAnimal`. Class `Omnivore` in turn implements the two interfaces `IHerbivore` and `ICarnivore`. In this case, class `Omnivore` has to implement methods of all interfaces, that is, `IAnimal`, `IHerbivore` as well as `ICarnivore` as shown in line1, line2, and line3.

Some advantages of interface are as follows:

- Interface allows simulating multiple inheritance of classes.
- Interface helps to avoid ambiguity between the methods of the different classes as seen in C++.
- Interface allows combining features of two or more interfaces such that a class needs to only implement the combined result.
- Interface allows name hiding. Since an interface cannot be instantiated, it helps to hide an inherited member name from any code outside the derived class.

After studying interfaces in such detail, a question may arise as to when a user can use interfaces?

The answer is that both Interfaces and Inheritance can be used to treat dissimilar objects collectively. For example, if there are two classes `Herbivore` and `Carnivore`, but there is some function that needs to handle them together, the user can:

- Create an `Animal` base class, which contains features common to both classes `Herbivore` and `Carnivore`, and can have `Herbivore` and `Carnivore` inherit from class `Animal`, or
- Create an `IAnimal` interface, and have both `Herbivore` and `Carnivore` classes implement the interface.

Now, whether to use the first or second approach depends on several factors. However, in general,

- If one is extending an existing class, then one can use inheritance. For example, if there is already a class called `Animal`, and it is required to distinguish between `Herbivore` and `Carnivore`, the user can apply inheritance.
- If one simply wants to treat different objects as the same, then one can use interfaces. For example, if there are already classes namely `Herbivore` and `Carnivore`, and it is required to manipulate them in a similar manner through a common functionality, without relating them, the user can use interface.

Interface is considered to be a contract between the class and the outside world which is enforced at compile time by the compiler. If a class implements an interface, all methods declared by that interface must be implemented in the class to successfully compile.

5.4 Multiple Inheritance and Constructors

When a class inherits from more than one class, the order of execution of constructors and initialization of their data fields becomes important. When the parent class constructors are no-argument constructors, they are invoked in the sequence in which the child class has inherited the parent classes. For example, if class `Omnivore` inherits class `Herbivore` first and then class `Carnivore`, the constructor of class `Herbivore` will be invoked, followed by constructor of class `Carnivore`, and finally, constructor of class `Omnivore`.

When the parent class has parameterized constructors, the user can handle this by invoking the parent class constructor from the constructor of the child class and pass arguments to the constructor as discussed in Code Snippet 1. Again, the invocation of parameterized or no-argument constructors also depends on the sequence in which the child class invokes them in its own constructor as shown in the C# code given in Code Snippet 9.

Code Snippet 9:

```
// class Omnivore inherits classes Herbivore and Carnivore
class Omnivore : public Herbivore, public Carnivore
{
   pubic:
     Omnivore : Carnivore(), Herbivore()   // line1
     {
          //…..
     }
}
```

In Code Snippet 9, even though the child class `Omnivore` inherits class `Herbivore` first and then class `Carnivore`, the constructor execution sequence will be `Carnivore()`, `Herbivore()`, and then `Omnivore()`. This is because, while invoking the constructors, the child class invokes class `Carnivore's` constructor first and then of class `Herbivore`.

Concepts

Summary

- Multiple Inheritance is a technique in which one class inherits its properties from more than one classes.
- Diamond Problem is a situation that arises when a class inherits from two parent classes each of which in turn inherit from a common super class.
- An interface is a type similar to a class that consists of only method declarations without any implementation.
- A class can implement more than one interface to simulate multiple inheritance.
- Invocation of constructors in multiple inheritance depends on the sequence in which the child class invokes them in its own constructor.
- Ambiguity of name is a condition in multiple inheritance where the base classes have a method with the same name and signature.

Check Your Progress

1. When a class derives from more than one parent class, it is called _____.

A)	Multiple Inheritance
B)	Multilevel Inheritance
C)	Single Inheritance
D)	Hybrid Inheritance

2. Chris is a developer in Web solutions Ltd. He has been asked to develop a code where a class can inherit more than one classes to implement a common functionality. Chris uses the language C++ to accomplish the task. He writes the code as shown:

```
class Base1
{
 protected;
  void display();
}
class Base2 : public Base1
{
};
class Base3 : public Base1
{
};
class Derived : public Base1, public Base2
{
  void main()
  {
      Derived d1 =new Derived();
      d1.display();
   }
};
```

Check Your Progress

The code written by Chris is not having the desired effect. Identify the problem that Chris is facing in the code.

A)	Locality problem
B)	Reference problem
C)	Diamond problem
D)	Pointer problem

3. Consider the following code in C++:

```
class Child : public Base1, public Base2, public Base3
{
   public:
     Child : Base3(), Base1(),Base2()
     {
          //…..
     }
}
```

Arrange the following options in the order of constructor execution?

A)	Base1()
B)	Base3()
C)	Base2()
D)	Child()

4. Which of the following options about an interface are true and which ones are false?

A)	Interface allows simulating multiple inheritance of classes.
B)	An interface can be instantiated.
C)	A class implementing an interface must implement all methods declared in the interface.
D)	An interface can have concrete methods, that is, methods with a body.

Check Your Progress

5. Identify the correct method of declaring an interface?

A)	`interface ISample`
	`{`
	`  private void display();`
	`}`

B)	`protected interface ISample`
	`{`
	`  void display();`
	`}`

C)	`interface ISample`
	`{`
	`  void display()`
	`  {`
	`  // statement1`
	`  }`
	`}`

D)	`interface ISample`
	`{`
	`  void display();`
	`}`

Try It Yourself

1. Crotus Inc. is a software development company in Philadelphia. The company employs developers and designers for software development. The employees can be permanent or contract based. Some employees come full time, whereas some are part time. There are several designations assigned to the employees such as Project manager, Project lead, Developer, and Designer. Diane works as a Web designer in the company. Identify the classes in the scenario, draw the inheritance hierarchy, and place Diane in the appropriate category in the inheritance hierarchy.

2. Consider the classes namely, `Vehicle`, `TwoWheeler`, `FourWheeler`, `Car`, and `Scooter`. There are two methods namely, `display (int vId, float vPrice)` and `bool isGeared()`. Identify the classes that can be converted to interfaces. Place the methods in the appropriate classes and interfaces. Draw a diagram to explain the scenario.

Session 6

Polymorphism

Objectives

At the end of this session, the student will be able to:

- *Explain Polymorphism*
- *List the different forms of Polymorphism*
- *Define Overloading and Overriding*
- *Define Polymorphic variable and Generics*
- *Explain Static and Dynamic Polymorphism*

6.1 Introduction

Diversity is a common feature of objects in nature. That is, objects in the world can appear and behave in a variety of forms in order to adapt to the changing environment or needs. This adaptation to changing environment helps the objects to survive during difficult times. Over centuries, the real world species have evolved by changing their size, shape, and other characteristics. These species can exhibit more than one form as and when required. That is to say they have 'many forms'. Therefore, they can be termed as 'poly-morphs'.

This session explains the concept of polymorphism which describes the way in which objects of classes can change their form and behavior as and when required. Finally, this session explains the different forms of polymorphism such as Overloading, Overriding, Polymorphic variable, and Generics.

6.2 Polymorphism

In Object-oriented programming, the term polymorphism means to assign a different usage or meaning to something in different contexts. The word polymorphism has a Greek origin where the word 'poly' means 'many' and 'morphos' means 'forms'. Thus, a polymorph is one that can appear or behave in more than one form. For example, a chameleon is an animal that changes its color to blend it with the environment. This it does to protect itself from its predators. Another example can be of a dog. When a dog smells a cat it will bark and run after it but when it smells food it will salivate and run to its bowl. Here, the dog's sense of smell is at work in both situations but action depends upon what it smells. Thus, a chameleon and a dog can be considered polymorphs.

Similarly, a human being can also be considered a polymorph. To understand this, consider a person named Chris. Chris is a doctor, a professor, a father of a child, and a son to his father.

Concepts

So, Chris can be considered a polymorph as he executes many roles in life and behaves in a different manner in different situations. As a doctor, his responsibility is to treat the patients. As a professor, he needs to teach his students, as a father, he has to take care of his child and as a son he has to look after his parents. Figure 6.1 shows a human being in different roles.

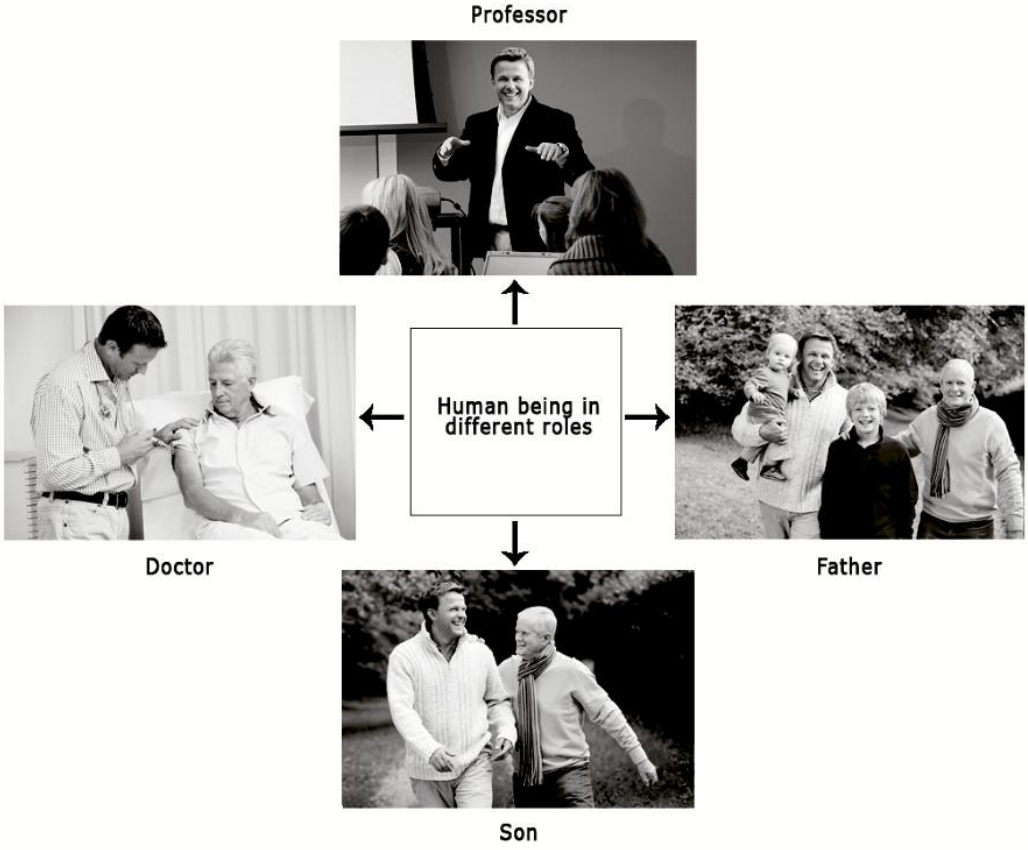


Figure 6.1: Human Being in Different Roles

Figure 6.1 shows how a human being exhibits polymorphism by playing various roles in life.

Likewise, in OOP, polymorphism allows an entity such a variable, a method, or an object to have more than one form. This enables code reusability and easy maintenance. Polymorphism is implemented in several ways in OOP such as overloading, overriding, polymorphic variable, and generics.

6.2.1 Overloading

Consider the situation when it is required to display the characteristics of a person such as `name`, `age`, and `address`, but all characteristics should not be displayed at the same time. Only those characteristics that are required by the object should be displayed. In such a case, it is difficult to have only one method to display all the features. Also, making methods with different names such as `displayName()`, `displayAge()`, and `displayAddress()` is a cumbersome task. In such a case, one would have to invoke each method separately even when one wants two or more of the characteristics to be displayed. Figure 6.2 shows an object `p1` of class `Person` accessing three methods namely, `displayName()`, `displayAge()`, and `displayAddress()` to display the `name`, `age`, and `address` of a person.

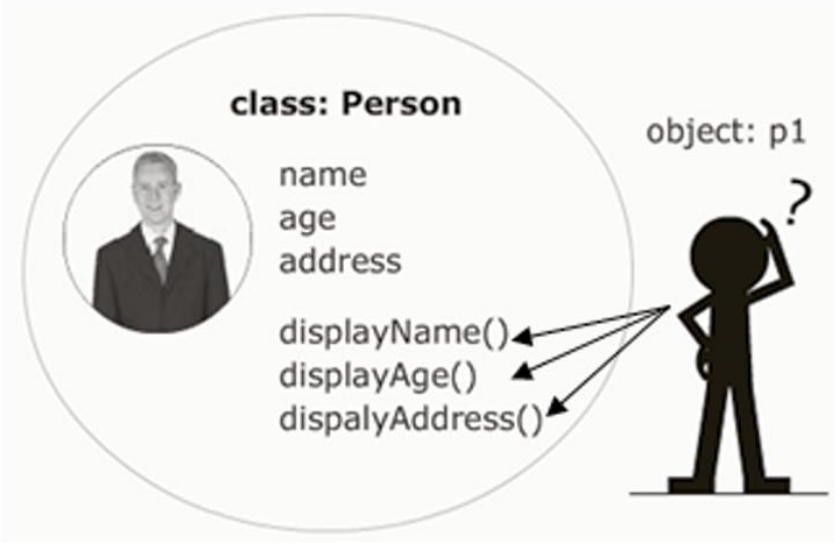


Figure 6.2: Class and Object

Figure 6.2 shows that to display three characteristics of a person, the object has to access three methods with different names. Instead, it would be useful to have a technique where a method with the same name can be used to display different values simply by changing the parameters of those methods to differentiate between them. OOP provides such a concept called method overloading.

Method Overloading is a technique in which a method with the same name can have several implementations by changing its signature. Here, changing the signature involves changing:

- the number of parameters,
- the type of parameters, or
- the sequence of parameters

Concepts

However, simply changing the return type or the names of the parameters does not make a method overloaded. To understand this better, consider the C# code given in Code Snippet 1.

Code Snippet 1:

```
class Person
{
    void display(string name)    // line1
   {
          // print name
   }
   void display(string name, int age)   // line2
   {
           // print name and age
    }
  void display(int age,  string name)    // line3
  {
        // print age and name
  }
   // not overloaded
   int display(int age, string name)  // line4
  {
      // print age and name
   }
  void display(string name, string address)  // line5
  {
          // print name and address
  }
    //   not overloaded
   void display(string pname, string addr)   // line6
   {
        // print address and name
   }
  void display(string name, int age, string addr)
   {
       // print name, age, and address
   }

  static void Main()
  {
```

```
    Person p1 = new Person();
    p1.display("Brian");      // line7
    p1.display("Brian",20);   // line8
    p1.display(20, "Brian");  // line9 - error
    p1.display("Brian", "Chicago"); // line10 - error
  }
}// end of class
```

Code Snippet 1 shows a class named `Person`. There are several `display()` methods in the class each with a different set of parameters. Now, the user has the flexibility to invoke the methods as per requirement. For example, if one only wishes to display the name, one can invoke the first method which takes only one parameter name as shown in line7. Similarly, if one wishes to print the name and age, one can invoke any of the methods of line2, line3, and line4 as shown in line8.

However, the method `display(int age, string name)` shown in line4 is not an overloaded method as its signature is similar to the `display()` method shown in line3. The only thing different is the return type. That is, `display()` method of line3 returns void. whereas `display()` method of line4 returns int. This does not make the `display()` method of line4 overloaded. Therefore, when the user invokes a `display()` method by passing the age first and then the name as shown in line9, the compiler issues an error because there is an ambiguity as to which `display()` method to invoke since both line3 and line4 `display()` methods take first parameter as `int` and second as `string`.

Similarly, the `display()` method at line6 that takes two strings cannot be considered overloaded. Its signature is same as the `display()`method at line5. The only thing changed is the names of the parameters. Therefore, when user invokes a `display()` method by passing two strings as shown in line10, the compiler issues an error as it does not know which `display()` method to invoke, that is, the one at line5 or line6.

6.2.2 Overriding

In general terms, to 'override' means to overrule or disobey. Consider a situation when a father asks the son to go into his room and read. When father says read, he means read books on education or syllabus related to school subjects such as Science or Mathematics. The child goes into the room and reads a book on comics. Thus, the child overrules or disobeys his father's command.

Concepts

The child implements the command in his own way disregarding what his parent has said. This is shown in figure 6.3.

Father asks Son to read Science Son disobeys and reads Comics

Figure 6.3: Son Overrules Father

Figure 6.3 shows the son disobeying father's instructions. In, OOP, this is known as method overriding. That is, a child class can override the method of a parent class and change the implementation of the method to suit its own requirement. This gives the child class liberty to use methods inherited from the parent class as needed. However, the child class cannot change the signature, that is, the name and parameters of the parent class method. It can only change the implementation of the parent class method. The C# code given in Code Snippet 2 shows method overriding.

Code Snippet 2:

```
class Father
{
    public virtual void read()    // line1
   {
       Console.WriteLine("Read Science ");
   }
}
  // class son inherits class father
 class Son : Father
 {
     // overridden method
    public override void read()    // line2
   {
       Console.WriteLine("Read Comics ");
   }
  static void Main()
  {
     Son s1 = new Son();
```

```
    s1.read();      // line3
  }
}// end of class

Output:
Read Comics
```

Code Snippet 2 shows a class named `Father` with a method named `read()`. The `read()` method in class `Father` displays the statement '**`Read Science`**'. Now, a class `Son`, inherits class `Father` and in turn inherits the method `read()`. However, the child class changes the implementation of the `read()` method to make it display '**`Read Comics`**' in C# of '**`Read Science`**'. That is, class `Son` overrides the class `Father` by changing the functionality of the `read()` method. In C#, to override a method, it should be marked '`virtual`' in the parent class as shown in line1 and it should prefixed with keyword 'override' in the child class as shown in line2. Now, when an object `s1` of the class `Son` is created, the statement `s1.read()` will invoke the class `Son`'s method and display '**`Read Comics`**' as shown in line3. Thus, functionality of parent class was overridden by the child class to suit its own requirement. This is known as method overriding.

6.2.3 Polymorphic Variable

From Code Snippet 2, it is clear that if you wish to invoke the `read()` method of the child class, you need to create an object of the class `Son` as follows:

```
Son s1 = new Son();
s1.read();
```

Similarly, to invoke the `read()` method of the parent, you have to make an object of class `Father` as follows:

```
Father f1 = new Father();
f1.read();
```

Now, suppose it is required to invoke methods of both parent as well as child class, it would not be necessary to create objects of both the classes, that is, parent as well as child class. Figure 6.4 shows two separate objects of class `Father` and class `Son` that is used to invoke the `read()` method of the respective classes.

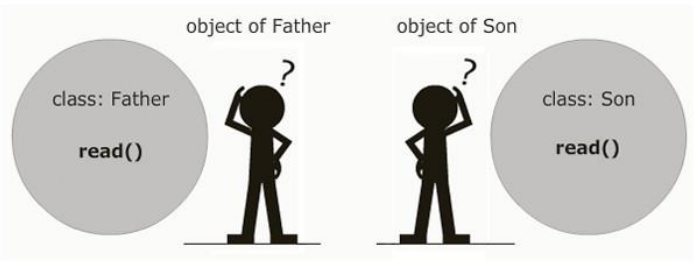


Figure 6.4: Classes and Objects

Concepts

Now, since child is already a type of parent, it should be possible to invoke the child class method using parent class object. OOP provides such a technique by which an object of parent class can be assigned reference to a child class. Such a variable is known as a polymorphic variable. Figure 6.5 shows a polymorphic variable.

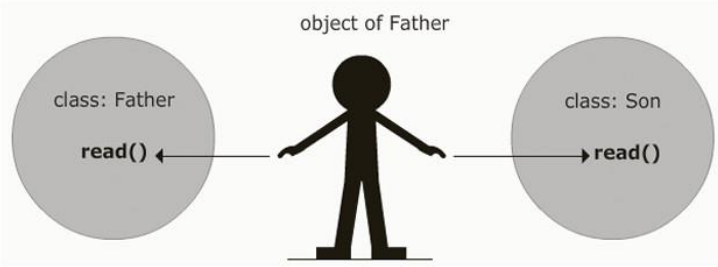


Figure 6.5: Polymorphic Variable

The C# code given in Code Snippet 3 shows the creation of polymorphic variable.

Code Snippet 3:

```
class Father
{
    public virtual void read()
   {
       Console.WriteLine("Read Science ");
   }
}
  // class son inherits class father
  class Son : Father
  {
     // overridden method
    public override void read()
   {
       Console.WriteLine("Read Comics ");
    }
    static void Main()
    {
     Father f1;       // line1
     f1 = new Father();   // line2
     f1.read();      // line3
     f1 = new Son();    // line4
```

```
    f1.read();      // line5
   }
}// end of class

Output:
 Read Science
 Read Comics
```

Code Snippet 3 shows declaration of object `f1` of class `Father` in line1. Initially, the object `f1` is assigned reference to its own class, that is, `Father` as shown in line2. When the statement `f1.read()` in line3 will execute, it will display '**`Read Science`**'. That is, the `read()` method of class `Father` will be invoked. Later, in line4, object `f1` is assigned the reference of class `Son`. Now, when the statement `f1.read()` of line5 is executed, it invokes the `read()` method of the class `Son` and displays '**`Read Comics`**' on the console. Thus, the same object `f1` is used to invoke both parent as well as child class `read()` methods.

This shows that the object `f1` exhibits polymorphic behavior, because `f1` is a variable that is declared as one type but holds a value of a different type. Therefore, object `f1` can be termed as a polymorphic variable.

6.2.4 Generics

The word 'generic' means 'general' or 'universal'. It refers to something which is generally applicable. Consider a situation where you wish to allow a user to enter a value of his choice and store it into a variable. In this case, you would require a variable that can store any kind of value supplied by the user at runtime. That is not possible because traditionally a variable has to have a type so that it can store a value. However, OOP provides a way of creating a general purpose variable or function whose type is decided based on the value passed by the user at runtime. A generic function or class is parameterized by values of unspecified type. Generic is also known as a Template. The type of value of a generic function or variable is decided later based on value supplied by user. Figure 6.6 shows an example of generics.

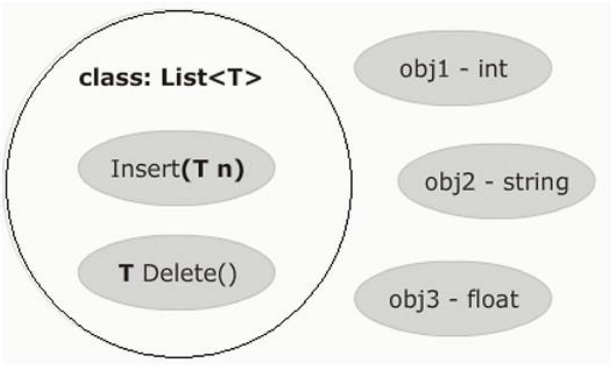


Figure 6.6: Generics

Figure 6.6 shows that based on the value supplied by the user while creating the object, the type of the variable n will be decided. It could be `int`, `float`, `string`, or even an `object` type.

The C# code given in Code Snippet 4 shows an example of generics.

Code Snippet 4:

```
public class List<T>  // line1
{
   public void Insert(T n)   // line2
   {...}
   public T Delete()    // line3
   {...}
 static void Main()
{
  List<int> myInt = new List<int>();   // line4
  myInt.Insert(10);    // line5
  myInt.Insert(20);
  int number = myInt.Delete();   // line6
}
```

Code Snippet 4 shows a class named `List` with two methods namely `Insert()` and `Delete()`. The symbol 'T' shown in line1, line2, and line3 are used to indicate type of the variable which will be supplied later. While creating the object `myInt` of class `List` in line4, the user supplies the value `<int>`. This is known as generics. This tells the compiler that the variable is going to hold an `integer` value as shown in line5. Similarly, the return type of method `Delete()` becomes an integer which can be then stored in an integer variable as shown in line6.

This way, by using generics, one can assign `<float>`, `<string>`, or any other class type to the object `myInt` based on the requirement and accordingly a list of that type will be created. Thus, class `List` becomes a general purpose class which can be used for any type.

6.3 Static and Dynamic Polymorphism

Polymorphism can also be categorized in another way based on the time at which the methods and variables are bound to their calls. The two classifications based on data and method call binding are termed as static and dynamic polymorphism.

- **Static Polymorphism**

  Static polymorphism is a technique where the method calls and data bindings are fixed at compile time. It is generally used with overloaded methods.

It refers to a component existing in different forms at a time. Static polymorphism involves binding of functions based on the number, type, and sequence of parameters. The types of parameters are assigned in the method declaration so that the method can be bound to calls at compile time. This form of binding is called early binding. The term early binding is used to denote that when the program is executed, the calls are already bound to the appropriate methods. The method call is resolved based on number, type, and sequence of parameters declared for each form of the method. For example, consider the following methods:

```
void display(string);

void display(int, float);
```

When the `display()` method is invoked, the values passed to it will determine which version of the method will be executed. This resolution is accomplished at compile time.

- **Dynamic Polymorphism**

Dynamic polymorphism is a technique where the method calls and data bindings are not fixed at compile time and are decided at runtime. It refers to an entity that changes its form as per the circumstances. It is generally used to refer to an object of a class that overrides a super class method or implements an interface. An object is said to exhibit dynamic polymorphism when its type is not fixed at compile time but is decided by the value passed by user at runtime. This form of binding is called late binding. This feature improves the flexibility of the program by allowing the appropriate method to be invoked, depending on the object type. For example, consider the classes shown in the C# code given in Code Snippet 5.

Code Snippet 5:

```
class Employee
{
    public virtual void display()
   {
       Console.WriteLine("Employee");
   }
}
 class FullTimeEmp : Emp
 {

    public override void display()
   {
       Console.WriteLine("Full Time Employee");
   }
}
```

Concepts

```
class Sample
{
   static void Main()
  {
     Employee e1;      // line1
     e1 = new FullTimeEmp();   // line2
     e1.display();    // line3
   }
}
```

Output:

```
 Full Time Employee
```

In Code Snippet 5, line1 shows the declaration of an object of class `Employee`. However, no reference has yet been assigned to it. Now, in line2, a reference of class `FullTimeEmp` is assigned to it. Therefore, a call to function `display()` at line3 will invoke the `display()` method of class `FullTimeEmp` and not of class `Employee`. This is called dynamic polymorphism where the type of the object is not fixed during compile time but decided at runtime.

- **Static Vs Dynamic Polymorphism**

  Static polymorphism is considered to be more efficient. whereas dynamic polymorphism is more flexible. The statically bound functions are bound to their calls at compile time. whereas dynamic function calls are bound to the functions at runtime based on the object type. This involves an additional overhead of searching the functions during runtime. On the other hand, no runtime search is required for functions that are statically bound.

Summary

- Polymorphism is a technique that allows an entity such as a variable, a method, or an object to have more than one form.
- Method Overloading is a technique in which a method with the same name can have several implementations by changing its signature.
- Method Overriding is a technique where a child class can change the implementation of the method inherited from the parent class to suit its own requirement.
- An object that is declared as one type but holds a value of a different type is known as a polymorphic variable.
- A generic function or variable is one whose type is not fixed during compilation but is decided by the value passed by the user at runtime.
- Static polymorphism is a technique where the function calls and data bindings are fixed at compile time.
- Dynamic polymorphism is a technique where the function calls and data bindings are not fixed at compile time and are decided at runtime.

Check Your Progress

1. Which of the following option is a technique in which a method with the same name can have several implementations by changing its signature?

A)	Overloading
B)	Overriding
C)	Inheritance
D)	Generics

2. Match the following description with their corresponding polymorphism type.

	Description		Polymorphism type
A)	Employee overrules Employer	**1)**	Generics
B)	Methods with same name but different parameters	**2)**	Polymorphic variable
C)	Used to assign type to a variable at runtimer	**3)**	Overloading
D)	An object that can hold reference of its child class	**4)**	Overriding

3. Consider the following C# code:

```
class Employee
{
    void display() { …}
}
  static void Main()
  {
     Employee e1;
     e1 = new Employee();
     e1.display();      // line1
     …….   // line2
     e1.display();      // line3
   }
```

In the code, suppose there are two classes `FullTimeEmp` and `PartTimeEmp` that inherit from class `Employee`. Both have `display()` methods overridden from the parent class `Employee`. Which the following options can be used at line2 to invoke the `display()` method

Check Your Progress

of class `PartTimeEmp`?

A)	`e1 = new FullTimeEmp();`
B)	`e1 = PartTimeEmp();`
C)	`e1 = new PartTimeEmp();`
D)	`e1 = new PartTimeEmp;`

4. Which of the following options about overloading are true and which ones are false?

A)	Overloading is a technique where a method with the same name can be used to display different values simply by changing the parameters.
B)	To overload a method, the number, type, and sequence of parameters need to be changed.
C)	Two methods with same name but different return type can be considered overloaded.
D)	Changing only the names of the parameters does not make a method overloaded.

Check Your Progress

5. Consider the following code:

```
class Parent
{
    public virtual void display()
   {
       Console.WriteLine("Study");
   }

 class Child : Parent
 {

    public override void display()
   {
       Console.WriteLine("Play");
   }
  static void Main()
  {
     Parent p1 = new Parent();
     p1.display();
   }
 }
```

What will be the output of the code?

A)	The code will throw compile time error.
B)	The code will throw runtime error.
C)	The code will run Successfully and print 'Study' on the console.
D)	The code will run Successfully and print 'Play' on the console.

Session 7

Overloading

Concepts

Objectives

At the end of this session, the student will be able to:

- *Explain Overloading*
- *List the different forms of Overloading*
- *Explain Method Overloading*
- *Explain Constructor Overloading*
- *Explain Operator Overloading*

7.1 Introduction

When someone looks up a dictionary, one will find words that have several meanings. That is, the same word can be used to convey a different message based on the context in which it is used. For example, consider the word 'bug'. Talking about a bug in medical terms means you are referring to an insect that probably caused the disease. Whereas, referring to it in technical terms, it means a defect in software. This means that the meaning of the word 'bug' changes with the context in which it is used. Similarly, in Object-oriented programming, the meaning or behavior of a function or method can be changed based on the context in which it is used.

This session explains the concept of overloading which describes the way in which a single function can be implemented in several ways. Finally, the session explains the different forms of overloading such as Method overloading, Constructor overloading, and Operator overloading.

7.2 Overloading

Consider the situation when it is required to add some values and the values can be of different types such as integers, decimals, or even strings. Also, at times more than two values may be required to be added at a time. In such a case, it is difficult to have only one method to add. Also, creating methods with different names such as `addInteger()`, `addFloat()`, and `addString()` is a cumbersome task. Figure 7.1 shows an object `c1` of class `Calc` accessing three different versions of `add()` methods namely, `addInteger()`, `addFloat()`, and `addString()` to add different types of values.

Concepts

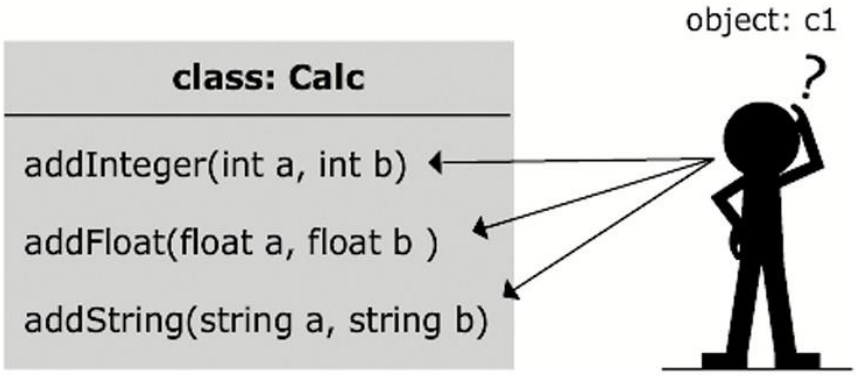


Figure 7.1: Class and Object

Figure 7.1 shows that to add different types of values, the object has to access three methods with different names. Instead, it would be useful to have a technique where several methods with the same name can be used to add different types of values. This can be done by changing the parameters of the methods to differentiate between them. OOP provides such a concept called Overloading.

Overloading is a technique in which a method with the same name can have several implementations by changing its signature. Here, changing the signature involves changing the:

- Number of parameters
- Type of parameters, or
- Sequence of parameters

7.3 Forms of Overloading

Overloading can be implemented in several ways in OOP such as Method overloading, Constructor overloading, and Operator overloading.

7.3.1 Method Overloading

Method overloading is the ability to define several methods with the same name but with different parameters. The compiler will be able to distinguish between the methods because of their different signatures. The modification of the signature implies changing the number, type, or sequence of parameters. However, simply changing the return type or the names of the parameters does not make a method overloaded. To understand this better, consider the example given in figure 7.2.

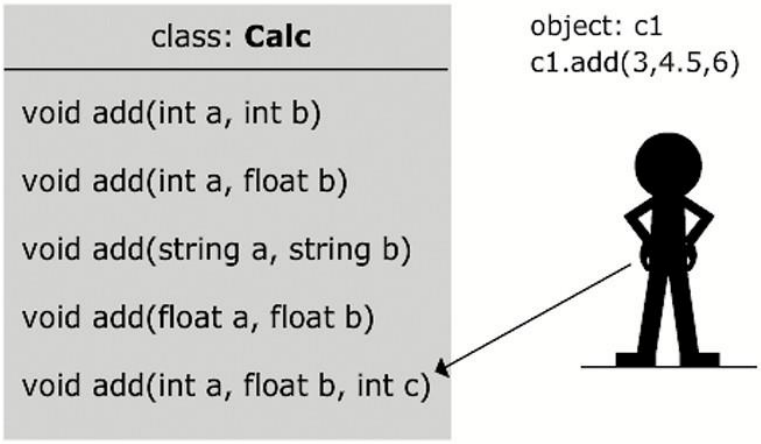


Figure 7.2: Method Overloading

Figure 7.2 shows an object `c1` of class `Calc` accessing the `add(int, float, int)` method by passing appropriate parameters. Now, there are several `add()` methods with different parameters, so that a different name for each method is not required.

The C# code given in Code Snippet 1 shows an example of method overloading.

Code Snippet 1:

```
class Calc
 {

   void add(int a, int b)   // line1
   {
            // add two integers
    }

  void add(int a,  float b)    // line2
  {
        // add integer and float
  }

   // not overloaded
   float add(int a, float b)  // line3
  {
```

Concepts

```
// add integer and float
   }
   void add (string a, string b)     // line4
   {
        // concatenate two strings
   }
  void add(float a, float b)  // line5
  {
          // add two float variables
  }

    //   not overloaded
   void add(float  x, float y)   // line6
   {
         // add two float variables
   }

   float add(int a, float b, int c)
  {
      // add two integers and a float
   }

  static void Main()
  {
     Calc c1 = new Calc();
     c1.add(10,10);      // line7
     c1.add(3, 3.5);  // line8 - error
     c1.add(20.5, 10.6);  // line9 - error
     c1.add("Brian", "Chicago");  // line10
  }
 }// end of class
```

Code Snippet 1 shows a class named `Calc`. There are several `add()` methods in the class each with a different set of parameters. Now, the user has the flexibility to invoke the methods as per the requirement. For example, if you wish to add two integers, you can use the `add()` method at line1 as shown in line7, whereas to add two strings you can use method at line4 as shown in line10. Similarly, to add an integer and a float, you can invoke any of the methods of line2 and line3 as shown in line8.

However, the method `add (int a, float b)` shown in line3 is not an overloaded method as its signature is similar to the `add()` method shown in line2.

The only thing different is the return type. That is, `add()` method of line2 returns `void` whereas `add()` method of line3 returns `float`. This does not make the `add()` method of line3 overloaded. Therefore, when the user invokes the `add()` method by passing an `integer` first and then a `float as` shown in line8, the compiler issues an error because there is an ambiguity as to which `add()` method to invoke since both add() methods at line2 and line3 take first parameter as `integer` and second as `float`.

Similarly, the `add()` method at line6 which takes two `float` parameters cannot be considered overloaded. Its signature is same as the `add()` method at line5. The only thing changed is the names of the parameters.

Therefore, when the user invokes the `add()` method by passing two `float` parameters as shown in line9, the compiler issues an error as it does not know which `add()` method to invoke, that is, the one at line5 or line6.

Thus, it is clear that a user cannot declare more than one method with the same name as well as same arguments as the compiler cannot differentiate between them. Since the compiler does not take into consideration the return type of the methods or the names of the parameters, one cannot declare two methods with the same name even if their return types are different. Such methods are not considered overloaded and will generate a compile time error when invoked.

The advantage of using overloading is that it allows methods with similar functionality to be used by a common name. Therefore, the name '`add`' here indicates a common action which will be performed on different types of values. All the user has to do is pass the required parameters and it is left to the compiler to invoke the correct version of the add method and execute it.

7.3.2 Constructor Overloading

A constructor is a special method that has the same name as the class name. It is used to initialize the data members of the class as soon as an object of the class is created. However, there might be a case where it is not required to initialize all data members of a class at a time. For example, consider the figure 7.3.

Figure 7.3: Class and Data Members

Concepts

Figure 7.3 shows a class `Person` with three data members namely, `name, age, and address`. Now, suppose the user wants to initialize all the data members inside the constructor with default values as soon as an object is created, it can be done as shown in the C# code given in Code Snippet 2.

Code Snippet 2:

```
class Person
{
     string name, address;
     int age;
   Person()   // line1
   {
          name = "Brian";
          address = "California";
          age = 23;

    }
  static void Main()
  {
     Person p1 = new Person();  // line2
  }
}// end of class
```

Code Snippet 2 shows that a class `Person` is created with a default or no-argument constructor as shown in line1. Inside this constructor, the data members `name, age`, and `address` have been initialized to default values. Now, when an object of the class Person is created as shown in line2, the default constructor of the class Person is invoked and the data members are initialized with the assigned values all at the same time.

However, one may not want to initialize all members together. One may want to initialize selected members only. In that case, a default constructor cannot be used. It is required to have a constructor with parameters. Again only one constructor may not suffice. The user will have to create multiple constructors with different arguments to fulfill the requirement. This means that the constructors will have to be overloaded as shown in the C# code given in Code Snippet 3.

Code Snippet 3:

```
class Person
{
     string name, address;
     int age;
```

```
   Person()    // line1
   {
          name = "Brian";
          address = "California";
          age = 23;
   }
   Person(string p_name, int p_age)    // line2
   {
          name = p_name;
          age = p_age;
    }

   Person(string p_name, string p_addr)    // line3
   {
          name = p_name;
          address = p_addr;
   }

  Person(string p_name, int p_age, string p_addr)  // line4
  {
          name = p_name;
          age = p_age;
          address = p_addr;
   }
    static void Main()
  {
      Person p1 = new Person("Brian", 23);  // line5
  }
 }// end of class
```

Code Snippet 3 shows class Person with several constructors all having different signatures as shown in line1, line2, line3, and line4. Now, the user has the flexibility to initialize any parameter as required. For example, in the code, the object of class `Person` is created at line5 with two parameters '`Brian`' and '`23`'. The compiler matches the parameters with the corresponding constructor which takes the string and integer parameters respectively. Thus, when line5 gets executed, the constructor at line2 is invoked and the values passed by the user are assigned to the respective data members.

Concepts

7.3.3 Operator Overloading

An operator is a symbol that is used to do some operation on values of different types. All programming languages provide different types of operators such as Arithmetic, Assignment, Relational, and Logical.

Table 7.1 shows the list of each of the commonly used operators.

Arithmetic operators	Assignment operator	Relational operators	Logical operators
+	=	==	&&
-		!=	||
*		<	!
/		>	
%		<=	
		>=	

Table 7.1: Types of Operators

Usually, the operators are used to work only with primitive types of data such as integer, float, string, and other such type. To use these operators with user-defined type such as a class, they have to be overloaded. Operator overloading is a technique used for extending the functionality of an existing operator so as to allow them to be used with user-defined types. For example, a '+' operator is normally used to add two numbers. With operator overloading, it can also be used to add strings and even objects of a class as shown in figure 7.4.

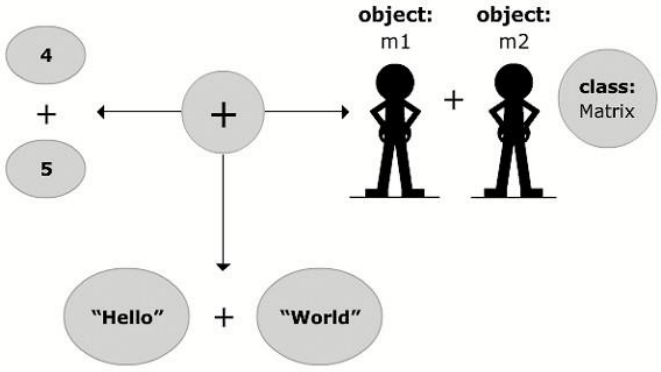


Figure 7.4: Overloading the '+' Operator

Figure 7.4 shows how the '+' operator can be used to add numbers, strings, and objects of a class.

Concepts

Operator overloading is a process of adding operator functionality to a user-defined type such as a class. This enables the user to define exactly how the operator behaves when used with his own class and other data types. It is used to add the values of two vectors or two complex numbers. It can also be used for multiplying matrices or non-arithmetic functions such as using the '+' operator to add a new item to a collection. It can also be used to combine contents of two arrays. Multiple overloaded versions of the operators can be created for processing different types of data similar to method overloading.

However, not all operators can be overloaded and for others, there are certain rules to be observed while overloading. The operators containing a 'dot' cannot be overloaded. For example, the member access operator '.', Inheritance operator ':', Property access operator '::', and Ternary operator '?' cannot be overloaded. Also, the relational operators have to be overloaded in pairs. For example, if '>' is overloaded then '<' must also be overloaded. Likewise, '>=', '<=' and '!=', '==' are overloaded in pairs.

The keyword '`operator`' is used to create an overloaded operator. An overloaded operator must be declared `static`. The syntax of declaring an overloaded operator is shown here.

Syntax:

```
public static <result-type> operator <operator> (op-type1 operand1, op-type2
operand2)
{
     // processing statements
}
```

where,

`result-type`: The data type of the return value.

`operator`: The symbol of the operator being overloaded.

`op-type1 and op-type2`: The data types of the two parameters.

`operand1 and operand2`: The two parameters to be passed to the method.

Though the syntax appears to be a rather complex declaration but it is quite simple. The declaration begins with keywords `public static` as all overloaded operators must be declared as such. Other scopes are not permitted for operator overloading and neither are non-static operators.

Here, result-type indicates the data type or class that is returned as a result of using the operator. Usually the result-type is the same as the class that it is being defined within. However, that is not mandatory and a data of a different type can also be returned.

Concepts

The keyword 'operator' is used to inform the compiler that the following operator symbol is a predefined operator rather than a normal method. This operator will be used to process the two operand parameters. Both parameters are prefixed with their data type, that is, 'op-type1' and 'op-type2'. At least one of these operands has to be of the same type as that of the containing class. For example, to overload the addition operator '+' for adding two strings, The C# code given in Code Snippet 4 shows an example of operator overloading.

Code Snippet 4:

```
public static string operator + (string str1, string str2)
{
     // processing statements
}
```

Code Snippet 4 shows overloading of operator '+' to add two strings which have been passed as parameters. The return-value of the function is also a string type.

Operator overloading is a very powerful but underused feature that helps the user to make the code much simpler. Adding operator overloading functionality to the class enables a user to:

- Convert data to and from one type to another
- Perform arithmetic and logical operations on a type with itself or other types

To understand the concept of operator overloading, consider a situation where a user wishes to add two values. The values could be of any type. The user can use the '+' operator and add the values or objects by using operator overloading. The C# code given in Code Snippet 5 shows an example of operator overloading.

Code Snippet 5:

```
using System;
public class Calc
{
  public int num;
  // parameterized constructor of class Calc
  public Calc(int num1)     //    line1
  {
     num = num1;
  }
 // declaring the overloaded operator +
public  static   int  operator +(Calc c1, Calc c2)     // line2
```

```
    {
      int sum = c1.num + c2.num;
      return sum;
    }

  public static void Main()
 {
     // Create two objects of class Calc and pass values for the variable
num

    Calc n1 = new Calc(3);     // line3
    Calc n2 = new Calc(4);     // line4
    // Add the two objects n1 and n2 of class Calc using the overloaded
operator +
      int ans = n1 + n2;     // line5
    // Display the two complex numbers
      Console.WriteLine("First number =  {0}  ", n1.num);
      Console.WriteLine("Second number = {0}  ", n2.num);
     // Display the addition
    Console.WriteLine("Addition = {0} ",+ ans); // line6

   }
}
```

Output:

```
First number = 3
Second number = 4
Addition = 7
```

Code Snippet 5 shows a class `Calc` with one integer type variable named `num`. It also has a constructor at line1 that initializes the data member with the value passed by user at runtime as shown in line3 and line4. The '+' operator has been overloaded at line2 with two arguments of type `Calc` and a result-type `integer`. This means that the operator `+` will add two objects of class `Calc` and return a value of type integer.

Inside the overloaded method of '+' operator, the real and imaginary parts of the `Calc` object `c1` and `c2` have been added. The method then returns an integer.

Concepts

Within the `main()` method, two objects of class `Calc` have been created as shown in line3 and line5 and values corresponding to the variable `num` are passed to the constructors. Next, at line`5`, the two objects are added using the overloaded '+' operator and the return value is stored in another variable `ans` of type `integer`. The sum is then printed at line6. In a similar manner, you can overload other operators such as '-', '*', '/', and other such operators.

Unlike C++ and C#, Java does not support operator overloading. It has been removed in order to avoid complexity in the code. It is not permitted in Java in order to prevent developers from misusing operators and creating convoluted implementations by assigning multiple meanings to the same operator that are hard for anyone to understand.

Summary

- Overloading is a technique in which a method with the same name can have several implementations by changing the number, type, or sequence of parameters.
- A method in which only the return type or the names of the parameters have been changed cannot be considered overloaded.
- A constructor is a special method that has the same name as the class name and is used to initialize the data members of the class.
- Operator overloading is a technique used for extending the functionality of an existing operator so as to allow them to be used with user-defined types such as a class.
- The keyword 'operator' is used to implement operator overloading in C#.
- Operator overloading enables a user to convert data to and from one type to another.
- Operator overloading enables a user to perform arithmetic and logical operations on a type with itself or other types.

Concepts

Check Your Progress

1. _____ is a technique in which a method with the same name can have several implementations by changing its signature.

A)	Overloading
B)	Overriding
C)	Inheritance
D)	Generics

2. Match the following code snippet with the corresponding overloading type.

	Code Snippet		Overloading Type
A)	`class Sample { Sample() {….} Sample(int a, int b) {…} }`	**1)**	Method overloading
B)	`class Sample { public static Sample - (Sample a, Sample b) {….} }`	**2)**	Constructor overloading
C)	`class Sample { display() {….} display(int a, int b) {…} }`	**3)**	Operator overloading

Check Your Progress

3. Consider the following C# code:

```
class Emp
{
      string name, address;
      int age;
    Emp() { …}
    Emp(string pname, string addr)    // line1
    {
              // print name and address
     }
     Emp(string addr, string name)    // line2
    {
             // print address and name
     }
  static void main()
{
     Emp e1 = new Emp("Robin","Chicago");    // line3
}
}
```

What will be the result of the code?

A)	The code will invoke constructor at line1 and print the values of name and address
B)	Compile time error
C)	No output
D)	The code will invoke constructor at line2 and print the values of name and address

4. Which of the following options regarding overloading are true and which ones are false?

A)	Overloading is a technique where a method with the same name can be used to display different values simply by changing the parameters.
B)	To overload a method, the number, type, and sequence of parameters need to be changed.
C)	Two methods with same name but different return type can be considered overloaded.
D)	Changing only the names of the parameters makes a method overloaded.

Check Your Progress

5. Which of the following operator cannot be overloaded?

A)	+
B)	*
C)	>
D)	?

6. Which of the following is the correct syntax for overloading the '+' operator?

A)	`public Object operator + (Object a, Object b)`
	`{`
	`  // processing statements`
	`}`

B)	`public static Object operator + (Object a, b)`
	`{`
	`  // processing statements`
	`}`

C)	`public static Object operator + (Object a, Object b)`
	`{`
	`  // processing statements`
	`}`

D)	`private static Object operator + (Object a, Object b)`
	`{`
	`  // processing statements`
	`}`

7. Match the following operators with their overloaded pair.

Operator		Pair	
A)	>	1)	>=
B)	!=	2)	<
C)	<=	3)	==

Session 8

Overriding

Objectives

At the end of this session, the student will be able to:

- *Explain Overriding*
- *Explain Abstract method and Pure virtual methods*
- *Explain Replacement and Refinement*
- *Differentiate between Overriding and Shadowing*
- *Differentiate between Overriding and Overloading*

8.1 Introduction

If you look around, you will find several objects that have multiple purposes. The size, shape, and other characteristics of the object remains the same, but its usage varies according to the requirement. For example, a pair of scissors can be used to cut paper, string, cloth, and even used during surgery. While cutting, the scissor remains the same in structure but the function of cutting is applied to different types of objects as required. Thus, the scissor exhibits polymorphism since its function can be applied in various ways. Similarly, in real world, the basic functionality or implementation of many objects can be changed without changing their structure.

This session explains the concept of Overriding which supports such multiple implementation of a component without changing its structure. Further, this session explains about different ways of implementing overriding such as Abstract methods and Pure Virtual methods. Finally, the session differentiates between the concept of Overriding, Shadowing, and Overloading.

8.2 Overriding

In general, the term 'override' means to overrule or disobey. Consider a situation when an employer Smith asks his employee Robert to work on projectX, but Robert is more interested in projectY. So, he goes to his desk and starts working on projectY. Thus, Robert overrules or disobeys his employer's command. He implements the command in his own way disregarding what his employer has said. This is shown in figure 8.1.

Figure 8.1: Employee Overrules Employer

Figure 8.1 shows the employee disobeying or overruling his employer's instructions. In, Object-oriented programming, such overruling of instructions is known as overriding. That is, a child class can override the method of a parent class and change the implementation of the method to suit its own requirement. This gives the child class liberty to use methods inherited from the parent class as needed. However, the child class cannot change the signature, that is, the name and parameters of the parent class method. It can only change the implementation of the parent class method. The C# code given in Code Snippet 1 shows an example of method overriding.

Code Snippet 1:

```
class Emp
{
    public virtual void display()    // line1
   {
       Console.WriteLine("Project X");
   }
}
 // class FullTimeEmp inherits class Emp
  class FullTimeEmp : Emp
  {
     // overridden method
    public override void display()    // line2
   {
       Console.WriteLine("Project Y");
   }
   static void Main()
```

```
  {

  FullTimeEmp e1 = new FullTimeEmp();
  e1.display();      // line3
  }
}// end of class


Output:
Project Y
```

Code Snippet 1 shows a class named Emp with a method named `display()`. The `display()` method in class `Emp` displays the statement '`Project X`'. Now, a class `FullTimeEmp`, inherits class `Emp` and in turn inherits the method `display()`. However, the child class changes the implementation of the `display()` method to make it display '`Project Y`' instead of '`Project X`'. That is, class `FullTimeEmp` overrides the class `Emp` by changing the functionality of the `display()` method. However, the child class is not allowed to change the signature of the parent class method.

In C#, to override a method, it should be marked '`virtual`' in the parent class as shown in line1 and it should be prefixed with the keyword '`override`' in the child class as shown in line2. Now, when an object e1 of the class `FullTimeEmp` is created, the statement `e1.display()` will invoke the class `FullTimeEmp`' `s` method and display '`Project Y`' as shown in line3. Thus, functionality of parent class was overridden by the child class to suit its own requirement.

However, it is also possible for the child class to invoke the parent class version of the overridden method along with its own version. This is implemented in different ways in different languages. For example, Java uses the keyword '`super`' whereas C# uses the keyword '`base`' to invoke a parent class version of an overridden method from the child class as shown in figure 8.2.

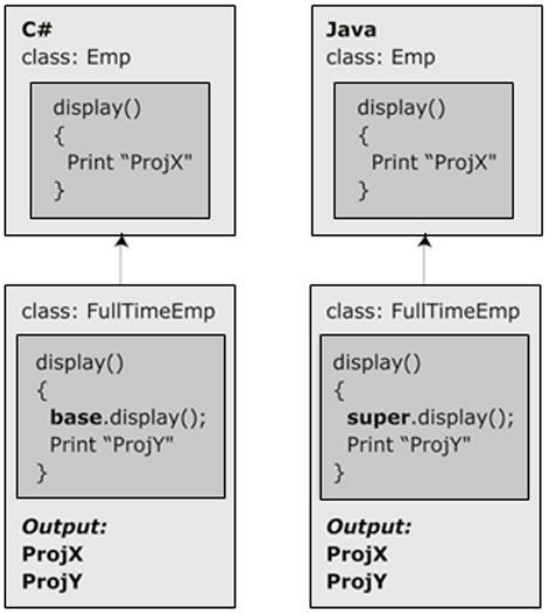


Figure 8.2: Calling Parent Class Method from Child Class Overridden Method

Figure 8.2 shows the use of keyword '`base`' in C# and '`super`' in Java for invoking the parent version of overridden method `display()` from the child class `display()` method. The statements '`base.display()`' and '`super.display()`' are used to make a call to the parent class `display()` method. Therefore, in both cases, when an object of `FullTimeEmp` class is created, the parent class `display()` method will be invoked first and then the rest of the child class method will be executed.

8.2.1 Replacement and Refinement

When the child class completely overrides the parent class method implementation, it is called Replacement. That is, the entire code of the parent class method is replaced by child class and the parent class version of the method is never executed when objects of child class are created as shown in Code Snippet 1.

However, when the child class includes the functionality of the parent class version of the overridden method within its own version, it is called Refinement. That is, the behavior of the parent is preserved and augmented by the child class as shown in figure 8.2.

8.2.2 Abstract Methods

Consider a situation where a user wants a common method in the parent class that should be used by all the child classes. However, the user wants the implementation of that method to be left to the child classes. Usually, you would think of using an interface since it allows only method declarations inside itself. What if you want some methods to be implemented in the parent class? Then, you cannot use an interface.

For example, consider a class named Shape which is inherited by sub-classes `Circle` and `Square`. Now, suppose the Shape class has a method `draw()`, it is difficult to decide what shape will be required to be drawn at runtime. Therefore, writing the body part of the `draw()` method inside the `Shape` class is of no use since it cannot be applied to all the child classes. It would be better to write the implementation of `draw()` method separately inside classes `Circle` and `Square`. In such a case, it is required to have a method that does not have an implementation inside the parent class. OOP provides such a concept called Abstract methods.

An abstract method is one which is declared inside parent class but not implemented there. Its definition is provided by the child class. Abstract methods are also called deferred methods since their implementation is deferred or delayed and is done in the child class. Deferred methods are created in different ways in different languages. Both C# and Java declare a deferred method using the keyword '`abstract`', whereas C++ declares the method as if assigning it the value zero along with the keyword '`virtual`' as shown in Table 8.1.

C#	Java	C++
abstract public void draw();	**abstract** public void draw();	**virtual** void draw() = 0;

Table 8.1: Syntax of Abstract Method in C#, Java, and C++

Table 8.1 shows syntax of writing an abstract method in C#, Java, and C++. The keyword '`abstract`' indicates that the method being modified has a missing implementation. The keyword '`abstract`' can be used with methods as well as classes.

When abstract modifier is used with a class declaration, it means that the class is intended only to be a base class of other classes and cannot be instantiated. Such a class is called an abstract class. Members marked abstract in an abstract class have to be implemented by the classes that inherit the abstract class.

The characteristics of abstract methods are as follows:

- An abstract method is considered implicitly virtual.
- Abstract methods are allowed only in abstract classes in C# and Java.
- An abstract method declaration provides no actual implementation and the method declaration simply ends with a semicolon.

Concepts

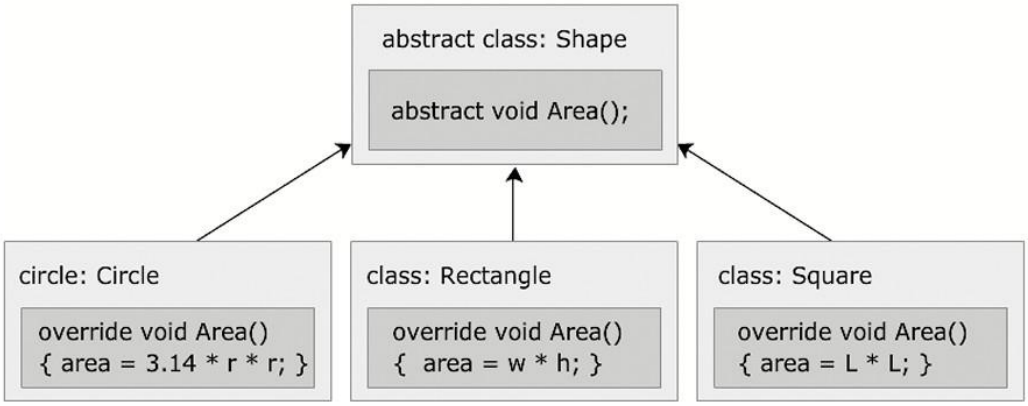


Figure 8.3: Abstract Class and Abstract Method

Figure 8.3 shows an abstract class `Shape` with an abstract method `Area()` which is used by the derived classes `Circle`, `Rectangle`, and `Square` to calculate the area of the various shapes. In the figure, `r` is radius, `w` is width, `h` is height, and `L` is length.

The C# code given in Code Snippet 2 shows an example of abstract class and abstract method.

Code Snippet 2:

```
abstract class Shape
{
    abstract public float Area();   // line1
}
// class Circle inherits Shape
class Circle : Shape
{
    float rad = 0;
    public Circle(float r)
    {
        rad = r;
    }
    public override float Area()      // line2
    {
        return 3.14F * rad * rad;
    }
static void Main()
```

```
  {
    Circle c1 = new Circle(2.5F);
    Console.WriteLine("Area is {0}", c1.Area());
  }
}

Output:
Area is 19.625
```

Code Snippet 2 shows an abstract class called Shape with a method called `Area()`. This method has been declared abstract as shown in line1. This means that the method cannot have a body inside the `Shape` class. Its implementation will be provided by the child class of class `Shape`. Now, the class `Circle` inherits class `Shape`. Therefore, class `Circle` will have to provide the implementation of the method `Area()` as shown in line2. As you can see that even though the method `Area()` in the `Shape` class was not prefixed with the keyword '`virtual`', still the keyword '`override`' has to be prefixed with the method `Area()` in the child class `Circle`. This is because method `Area()` in the Shape class is already prefixed with the keyword 'abstract' which makes it implicitly virtual. Therefore, it is not required to prefix it with '`virtual`' keyword as in normal overriding. Similarly, the user can inherit other classes such as `Square`, `Triangle`, and `Rectangle` from the `Shape` class and override the abstract method `Area()` to calculate the area of the respective shapes.

The details mentioned so far about the abstract class and abstract method may give rise to a question as to why an interface cannot be used for the same problem? At a superficial level, both abstract class and interface might look the same in structure and functionality. Both cannot be instantiated, both contain abstract methods and both have to be inherited or implemented by a child class. However, there are several differences in the implementation of the interface and abstract class. Table 8.2 lists the differences between the abstract class and interface.

Abstract Class	Interface
Can contain methods with a body	Cannot contain methods with a body
Cannot simulate multiple inheritance	Can simulate multiple inheritance
Child class should be a type of parent	Child class may not be a type of interface
Abstract class can have private members	All members of interface are implicitly public
Can have fields and constants	Cannot have fields and constants

Table 8.2: Difference between Abstract Class and Interface

Thus, if you simply want to apply a common functionality across classes without creating any relationship between them, you can use interface. If the classes also need to be related then you can use abstract class.

Concepts

8.2.3 Pure Virtual Functions

The deferred methods of C# and Java are referred to as pure virtual functions in C++. A pure virtual function in C++ is declared as shown in Code Snippet 3.

Code Snippet 3:

```
class Shape
{
     public:
      virtual void Area() = 0;
}
```

Code Snippet 3 shows a class Shape in C++ with a pure virtual function named `Area()`. The '`= 0`' at the end of the function declaration is used to indicate that the function has no implementation. In C++, a class having a pure virtual function is considered an abstract class. C++ provides different semantics for abstract classes. Similar to C# and Java, an abstract class in C++ also cannot have any instances. If an attempt is made to create an object, the compiler will issue an error. For example:

```
Shape s1;     // - error
Shape* s1 = new Shape; // - error
```

However, this constraint does not prevent the user from declaring pointers and references to abstract classes. The compiler will allow them to refer to objects of concrete classes which have used the abstract class somewhere in their inheritance hierarchy. This is shown in the C++ code given in Code Snippet 4.

Code Snippet 4:

```
class Circle : public Shape
{
      public:
        Circle() {}
        Circle(float r)
        {
            rad=r;
        }
         void Area()
         {
           float cArea = 3.14 * rad * rad;
           cout << "Area of Circle is "<<cArea<<endl;
```

```
        }
      private:
        float rad;
};
void main()
{
  Shape *s1 = new Circle(2.5);
  s1->Area();
  Shape &s2 = *(new Circle(2));
  s2.Area();
}

Output:
Area of Circle is 19.625
Area of Circle is 12.56
```

In Code Snippet 4, `s1` is a pointer variable. whereas `s2` is a reference created using the '`&`' operator. This means `s1` holds the address of the object of class `Circle` whereas `s2` holds the object itself.

Such pointers and references become useful for taking advantage of the polymorphic attributes of abstract classes. For example:

```
s1->Area();  // invokes Area() method of Circle using pointer notation
s2.Area();  //invokes Area() method of Circle using member access
              operator
```

The pointer s1 of class `Shape` is used to invoke the method `Area()` of the abstract class `Circle` using the '`->`' operator. Similarly, the reference variable `s2` is used to invoke `Area()` method of class `Circle` using '.' operator.

In C++, including a pure virtual function in a class is the only way to inform the compiler that the class is abstract. This is often considered to be a limitation by developers.

8.2.4 Overriding versus Shadowing

Due to superficial syntactic similarities, it is easy to confuse the mechanism of overriding and the related programming language concept of shadowing. Shadowing is a technique in which a variable or method declared in one scope hides a variable or method with the same name in another scope. An example of shadowing is shown in Code Snippet 5.

Concepts

Code Snippet 5:

```
class Display
{
  private int a;    // an instance variable --- line1

  public void show(int a) //  'a' shadows instance variable  - line2
  {
    int b = a + 1;
    while(b>3)
    {
     int a = 1;   // local variable shadows parameter 'a'    - line3
     b = b - a;
    }
  }
}
```

In Code Snippet 5, there are three different variables named 'a'. The first one is an instance variable shown in line1 which is accessible from any method defined in the class. In the method `show()` at line2, there is another parameter defined with the same name. This parameter hides or shadows access to the instance variable. This means that within the method, the name 'a', is matched to the parameter and not to the instance variable. However, to access the instance variable within the method you can use `this.x`, where keyword '`this`' indicates the current object. Within the while loop, another variable with the name 'a' has been declared. This is a local variable and it in turn shadows access to the parameter 'a'. It is not advisable to use variables in such a manner. However, the language definition does not prohibit it either.

Similarly, languages such as C++ which require an explicit indication for overriding will lead to shadowing if no `keyword` is provided. For example, if the '`virtual`' keyword is not used in C++ code while overriding a method, it will lead to shadowing of the method in the child class as shown in the C++ code given in Code Snippet 6.

Code Snippet 6:

```
class Shape
{
  public:
    void draw() //   note that virtual keyword is not used   -- line1
    {
       cout << "Draw any shape";
    }
};
```

```
class Circle : public Shape
{
      public:
         void draw()  //   line2
         {
           cout << "Draw Circle";
         }
};
void Main()
{
  Shape *s1 = new Shape();
  s1->draw();          //  line3
  Circle *c1 = new Circle();
  c1->draw();         //    line4
  //   assigning reference of Circle to Shape
  s1 = c1;       //   line5
  s1->draw();    // line6
}

Output:
Draw any shape
Draw Circle
Draw any shape  ---- ???
```

Code Snippet 6 shows a class `Shape` with a method called `draw()`. The `draw()` method is not declared with the keyword '`virtual`'. Class `Circle` inherits class `Shape` and changes the implementation of the method `draw()` to display something different as shown in line2. Now, when an object `s1` of class `Shape` is created and method `draw()` is invoked in line3, the output is '**`Draw any shape`**' as expected. Similarly, on creating object `c1` of class `Circle`, a call to `draw()` method prints '**`Draw Circle`**', again as expected.

When the reference of `c1` is assigned to s1 in line5 and the method `draw()` is invoked using `s1->draw()` in line6, the output is again '**`Draw any shape`**' and not '**`Draw Circle`**'. That is, even though the object of class `Shape` was assigned reference of class `Circle`, it did not resolve the reference. This is because, the method `draw()` of child class got shadowed by that of parent class as the parent class method was not declared '`virtual`'.

Therefore, since overriding is resolved at runtime, the binding of a variable happens late. On the other hand, shadowing is resolved at compile-time just like overloading. This means that the call to the method `draw()` by object `s1` is fixed at compile time. It will not vary even though reference of `c1` is assigned to `s1` because method `draw()` is not declared virtual in the parent class.

Concepts

8.2.5 Overriding versus Overloading

In OOP, both method overloading and method overriding are used to implement polymorphism. However, there are several differences between both the concepts that are shown in Table 8.3.

Method Overriding	Method Overloading
Classes in which overridden methods are used must have a parent-child relationship	Parent-child relationship is not required for overloading
For overriding, the method signature cannot be changed	For overloading, the method signature must be changed
Overridden methods can be combined to perform the actions of both parent and child class together	Overloaded methods are always executed separately
Overriding is resolved at runtime	Overloading is resolved at compile time

Table 8.3: Difference between Overriding and Overloading

Thus, to summarize the various concepts, you can cite the following differences between the different terms in polymorphism:

- In overriding, the signatures of both methods, that is, of parent and child classes are same. Also, the overridden method is declared as `virtual` in the parent class.
- In shadowing, the signatures of both methods, that is, of parent and child classes are same but the method in parent class is not declared as virtual.
- In redefinition, the name of the methods in parent and child class remains the same but signature of the child class method differs from that of the parent class. This is shown in the C# code given in Code Snippet 7.

Code Snippet 7:

```
class Emp
{
     public void display()
     {
          Console.WriteLine("Project X");
     }
}
class FullTimeEmp : Emp
{
     public void display(string projname)
     {
         Console.WriteLine("Project "+ projname);
     }
}
```

Code Snippet 7 shows a class `FullTimeEmp` that inherits class Emp. Class Emp has a method named display without any parameters. Class `FullTimeEmp` also has a method with the same name but with one string parameter. Thus, the child class uses the same method name as the parent class but changes the signature. This is known as redefinition.

- In overloading also the names of all methods remain the same and signature is changed. There is no parent-child relationship in overloading.

Summary

- Method overriding is a technique in which a child class can override the method of a parent class and change the implementation of the method to suit its own requirement.
- Replacement is a condition in which the child class completely overrides the parent class method implementation.
- Refinement is a condition in which the child class includes the functionality of the parent class version of the overridden method within its own version.
- An abstract method is one which is declared inside parent class but implemented in the child class.
- An abstract class is intended only to be a base class of other classes and cannot be instantiated.
- Shadowing is a technique in which a variable or method declared in one scope hides a variable or method with the same name in another scope.
- Redefinition is a technique in which the name of the methods remains the same but signature of the child class method differs from that of the parent class.

Check Your Progress

1. _____ is a technique in which child class completely overrides the parent class method implementation.

A)	Replacement
B)	Refinement
C)	Redefinition
D)	Overloading

2. Match the following concept with the corresponding description

	Concept		Description
A)	Refinement	**1)**	The name of the methods remains the same but signature of the child class method differs from that of the parent class
B)	Shadowing	**2)**	The child class includes the functionality of the parent class version of the overridden method within its own version
C)	Redefinition	**3)**	A technique in which a variable or method declared in one scope hides a variable or method with the same name in another scope

3. Which of the following options is not a similarity between abstract class and interface?

A)	Both cannot be instantiated
B)	Both contain abstract methods
C)	Both have to be inherited or implemented by a child class
D)	Both allow methods with a body

4. Which of the following options about an abstract method are true?

A)	An abstract method is considered implicitly virtual.
B)	Abstract methods are allowed in non-abstract classes.
C)	An abstract method declaration can provide implementation.
D)	The keyword 'abstract' indicates that the method being modified has a missing implementation.

Check Your Progress

5. Which of the following is the correct syntax for creating an abstract method?

A)	`abstract private void draw();`
B)	`public abstract void draw();`
C)	`void draw();`
D)	`abstract public void draw();`

6. Consider the following code snippet:

```
class Father
{
    public virtual void read()
    {
       Console.WriteLine("Study");
    }
}
class Son : Father
{

   public override void read()
   {
       Console.WriteLine("Play");
   }
   static void Main()
   {
     Son s1 = new Son();
     s1.read();
   }
 }
```

What will be the output of the code?

A)	The code will print 'Study' on the console.
B)	The code will throw runtime error.
C)	The code will print 'Play' on the console.
D)	The code will throw compile time error.

Check Your Progress

7. Match the following code snippets with their corresponding concept?

Code Snippet		Concept	

A)	`class Parent`
	`{`
	`public void display()`
	`{`
	`Console.WriteLine("Study");`
	`}`
	`}`
	`class Child : Parent`
	`{`
	`public void display()`
	`{`
	`Console.WriteLine("Play");`
	`}`
	`}`
		1)	Refinement

B)	`class Parent`
	`{`
	`public void display()`
	`{`
	`Console.WriteLine("Study");`
	`}`
	`}`
	`class Child : Parent`
	`{`
	`public void display(string name)`
	`{`
	`Console.WriteLine("Play", + name);`
	`}`
	`}`
		2)	Replacement

Check Your Progress

	Code Snippet		Concept

C)	`class Parent`
	`{`
	`  public void display()`
	`  {`
	`  Console.WriteLine("Study");`
	`  }`
	`}`
	`class Child : Parent`
	`{`
	`  public void display()`
	`  {`
	`  base.display();`
	`  Console.WriteLine("Play");`
	`  }`
	`}`
		3)	Redefinition

Session 9

Polymorphic Variable

Objectives

At the end of this session, the student will be able to:

- *Explain polymorphic variable*
- *List types of polymorphic variables*
- *Explain simple polymorphic variable*
- *Explain pseudo-variable*
- *Explain Reverse polymorphism*

9.1 Introduction

In Object-oriented programming, a class can extend another class using inheritance. Due to inheritance, all exposed properties of the parent class are derived by the child class. The child class also overrides the methods of the parent class. To access the overridden methods of a child class, usually a user creates an object of the child class. However, child class is a type of parent, it should be possible to access the child class method using the object of parent class. OOP provides such a concept called polymorphic variable.

This session explains the concept of polymorphic variable. Further, this session explains about the different forms of polymorphic variables. Finally, the session explains the concept of Reverse polymorphism.

9.2 Polymorphic Variable

A polymorphic variable is one that can hold reference of more than one type of object. Consider a situation where in a class inherits another class and overrides the parent class method. Now, to access the overridden method, the user would have to write a C# code as shown in Code Snippet 1.

Code Snippet 1:

```
class Father
{
   public virtual void study()   // line1
   {
      Console.WriteLine("Read Science");
   }
}
// class son inherits class father
class Son : Father
{

  // overridden method
  public override void study() // line2
  {
     Console.WriteLine("Read Comics ");
  }
  static void Main()
  {
    Son s1 = new Son();
    s1.study();     // line3
  }
}// end of class


Output:
Read Comics
```

In Code Snippet 1, the object of class `Son` is used to invoke the method `study()` of `Son` class as shown in line3. Now, if you want to invoke the `study()` method of the parent, you have to make an object of class `Father` and invoke the method as shown in Code Snippet 2.

Code Snippet 2:

```
Father f1 = new Father();
f1.study();
```

Now, suppose it is required to invoke methods of both parent as well as child class, it would be required to create objects of both the classes, that is, parent as well as child class.

Figure 9.1 shows two separate objects of class `Father` and class `Son` that are used to invoke the `study()` methods of the respective classes.

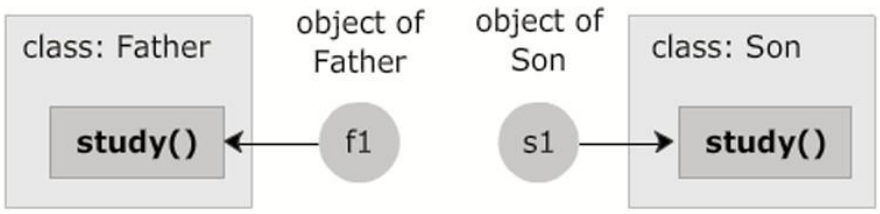


Figure 9.1: Classes and Objects

Now, since child class is a type of parent class, OOP provides a technique by which an object of parent class can be assigned reference to a child class. Such a variable is known as a polymorphic variable. Figure 9.2 shows a polymorphic variable.

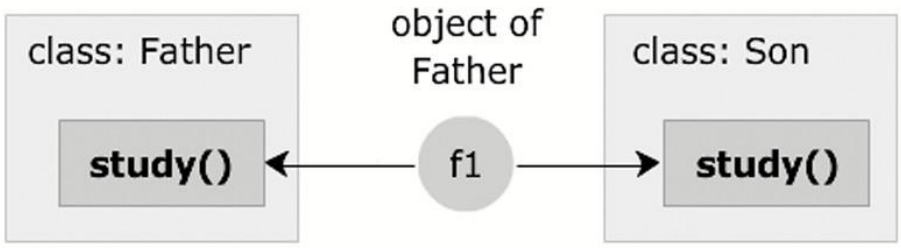


Figure 9.2: Polymorphic Variable

Figure 9.2 shows an object `f1` of class `Father` which is used to access both parent as well as child class `study()` methods.

9.3 Types of Polymorphic Variables

Polymorphic variables can have various implementations such as Simple polymorphic variable and Pseudo variable.

9.3.1 Simple Polymorphic Variable

Figure 9.2 shows an example of a polymorphic variable `f1` which can reference the parent as well as the child class. Similarly, consider a class `Emp` that has two sub-classes namely `FullTimeEmp` and `PartTimeEmp`. Now, the class `Emp` has a method called `display()` which is overridden in `FullTimeEmp` and `PartTimeEmp` classes. A polymorphic variable can be created that can reference the `display()` method of all the three classes as required.

Concepts

The C# code given in Code Snippet 3 shows the creation of polymorphic variable.

Code Snippet 3:

```
class Emp
{
  public virtual void display()
  {
    Console.WriteLine("Class Emp");
  }
}

// class FullTimeEmp inherits class Emp
class FullTimeEmp : Emp
{
 // overridden method
   public override void display()
   {
      Console.WriteLine("Class FullTimeEmp");
   }
}

// class PartTimeEmp inherits class Emp
class PartTimeEmp : Emp
{
 // overridden method
  public override void display()
  {
      Console.WriteLine("Class PartTimeEmp");
  }

  static void Main()
  {
    Emp e1 = new FullTimeEmp();  // polymorphic variable e1
    e1.display();  // line3
   }
}// end of class
```

```
Output:
Class FullTimeEmp
```

Code Snippet 3 shows declaration of three classes namely, `Emp`, `FullTimeEmp`, and `PartTimeEmp`. The class `Emp` has a method named `display()`. Classes `FullTimeEmp` and `PartTimeEmp` inherit from `Emp` and override the method `display()`. Within the `Main()` method, an object `e1` of class `Emp` is created. However, the object `e1` is assigned the reference of class `FullTimeEmp` instead of `Emp`. The statement '`new FullTimeEmp`' indicates a reference of class `FullTimeEmp`. Now, a call to the `display()` method using object `e1` will invoke the method `display()` of class `FullTimeEmp`.

This means, the object `e1` can hold a reference to class `FullTimeEmp` even though it is an object of class `Emp`. That is, `e1` is a variable that is declared as one type but holds a value of a different type. That is, it exhibits more than one form. Thus, variable `e1` can be called a polymorphic variable.

9.3.2 Pseudo-variable

It might sound ironic but the most common place where a polymorphic variable is used is a place where it is not visible at all. It is similar to an invisible receiver that works inside a method. Such a variable is normally not declared and therefore, it is called a pseudo-variable or the receiver variable. It has different names in various languages such as '`this`' in C++, C# and Java, '`self`' in Small talk, and '`current`' in a language called Eiffel.

This variable is used as a way of accessing data members as well as to act as receiver when methods are passed to 'oneself'. In either case, usually the variable is not explicitly declared or used in the code. However, explicit usage of the variable is not prohibited either. For example, consider the C# code given in Code Snippet 4.

Code Snippet 4:

```
class Sample
{
  private int val;

  public void calc(int a)
 {
     val = a * 3;
     add( a + 10);
 }
```

Concepts

```
 private void add(int b)
 {
    Console.WriteLine("Value is " + (val + b));
 }
}
```

Code Snippet 4 shows the use of the private variable 'val' in methods calc() and add() internally without any object. Similarly, the call to private method add() within method calc() is equally ambiguous. This is possible because the disambiguation comes from the implicit use of the pseudo-variable 'this'. In fact, the code given in Code Snippet 4 can be rewritten in an unambiguous manner as shown in the C# code given in Code Snippet 5.

Code Snippet 5:

```
class Sample
{
  private int val;

 public void calc(int a)
 {
     this.val = a * 3;      // line1
     this.add( a + 10);    // line2
 }

 private void add(int b)
 {
    Console.WriteLine("Value is " + (this.val + b));  // line3
 }
}
```

Code Snippet 5 shows the use of the pseudo-variable 'this' at line1, line2, and line3 to remove ambiguity in the code while accessing the members of the class from within itself.

Another use of the pseudo-variable is within the constructor or any method that is used for variable initialization and assignment. In such methods, the pseudo-variable eliminates the need to give different names to the method parameters from that of the actual parameters. Consider the C# code given in Code Snippet 6.

Code Snippet 6:

```
class Sample
{
  private int val;

  Sample(int val1)
  {
     val = val1;   // line1
  }

  public void setVal(int val2)
  {
      val = val2;     // line2
  }

  static void Main()
  {
       Sample s1 = new Sample(10);   // line3
  }
}
```

Code Snippet 6 shows the use of different parameter names 'val1' and 'val2' in the constructor and `setVal()` method of class `Sample`. This is no more required since the pseudo-variable can be used instead of using different names for the parameters. This is shown in the C# code given in Code Snippet 7.

Code Snippet 7:

```
class Sample
{
  private int val;

  Sample(int val)
  {
   // using 'this' to resolve name ambiguity

   this.val = val;   // line 1
  }
```

Concepts

```
  public void setVal(int val)
  {
    // using 'this' to resolve name ambiguity

    this.val = val;     // line 2
  }

  static void Main()
  {
       Sample s1 = new Sample(10);    // line 3
  }
}
```

In Code Snippet 7, class `Sample` consists of a variable named '`val`'. A constructor is used to initialize the value of the variable. However, the parameter of the constructor also has the same name, that is, '`val`'. Now, in normal cases, when a call to the constructor is made as shown in line 3, the similar names would cause a compile time error if one writes '`val=val`' inside the constructor. To resolve this issue, one can make use of the pseudo-variable 'this' as shown in line 1. By writing '`this.val`' for the instance variable, the compiler understands that the variable '`val`' on the left side of the assignment operator '`=`' is the instance variable, whereas the one on the right side is the parameter of the constructor. This removes the ambiguity and the code executes successfully.

Similarly, even in a method especially a Mutator, the pseudo-variable can be used to resolve name ambiguity as shown in line 2.

9.4 Reverse Polymorphism

Reverse polymorphism is a technique by which one can undo the substitution done in a polymorphic variable. In Code Snippet 3, the object `e1` of class `Emp` was assigned the reference of its child class `FullTimeEmp` which made object `e1` a polymorphic variable. That is, the object `e1` was declared as type parent but was actually holding the reference of the child class. Then, it should be possible to assign the object `e1` again to a variable of type child class. This is shown in Code Snippet 8.

Code Snippet 8:

```
Emp e1 = new FullTimeEmp(); // assign child class reference to parent class

FullTimeEmp e2 = (FullTimeEmp)e1; // cast the parent object back to child
type
```

In Code Snippet 8, first the reference of child class `FullTimeEmp` is assigned to the object `e1` of parent class `Emp`. Next, the object `e1` is casted back into child class `FullTimeEmp` and assigned to an object `e2` of class `FullTimeEmp`. Since it appears as undoing the polymorphic assignment, this process is termed as reverse polymorphism.

There are two problems that can arise from such an operation:

- **Problem of identification**: When casting a parent class object back to child class, it is necessary that the receiver object should be same as the child class to which the parent was referencing. Therefore, it is required to have some mode of identification of which child class object the parent was referencing.

- **Problem of assignment**: The assignment of the parent class object to a child class will fail if the identification of the parent class object has not been done correctly. This is shown in figure 9.3.

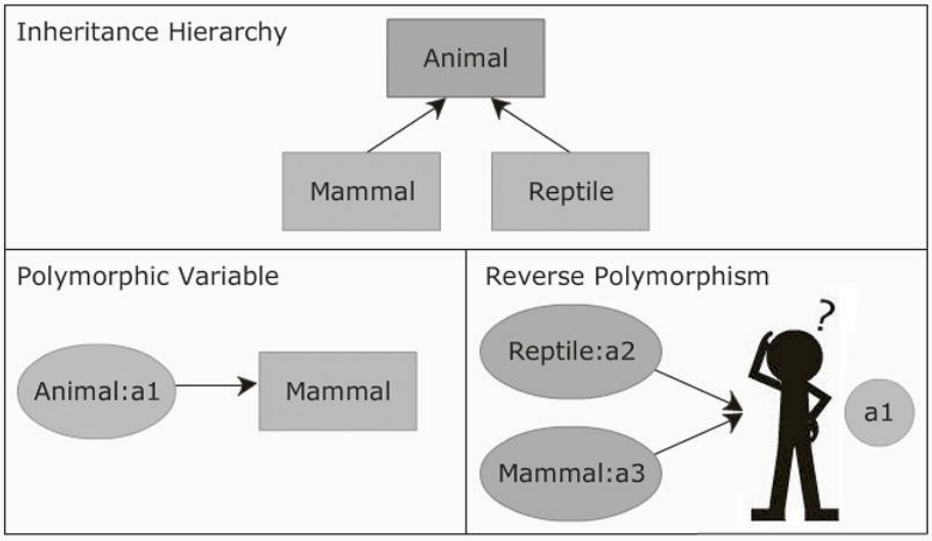


Figure 9.3: Identification and Assignment Problem

Figure 9.3 shows the classes `Reptile` and `Mammal` inheriting the `Animal` class. The object `a1` of class `Animal` is assigned the reference of class `Mammal` so that it becomes a polymorphic variable. Now, in order to do Reverse polymorphism, it is necessary to identify which child class object `a1` is referencing. Unless it is not done, it is difficult to decide which class the object `a1` should be assigned to. Since there are two child classes `Reptile` and `Mammal`, it is possible that reverse polymorphism may lead to an unsafe cast. This is shown in the C# code given in Code Snippet 9.

Concepts

Code Snippet 9:

```
class Animal
{
    public virtual void Move()
    {
      Console.WriteLine("Moving");
    }
}
class Mammal : Animal
{ }

class Reptile : Animal
{ }

class CastUnsafe
{
    public void Cast(Animal a1)
    {
        // Throws InvalidCastException at runtime
        // as Mammal cannot be converted to Reptile.

        Reptile r1 = (Reptile)a1;    // line1
    }

    static void Main()
    {
        CastUnsafe c1 = new CastUnsafe();
        Animal a1 = new Mammal();
        c1.Cast(a1);    // line2
    }
}
```

Code Snippet 9 shows three classes namely, `Animal`, `Mammal`, and `Reptile`. `Mammal` and `Reptile` are child classes of class `Animal`. Another class `CastUnsafe` consists of a method named `Cast()` which takes the object of class `Animal` as a parameter. Within the method, the object of `Animal` class is casted back into `Reptile` as shown in line1. In the `Main()` method, an object of `CastUnsafe` class is created and the method `Cast()` is invoked as shown in line2. Note that the reference passed inside the method is that of class `Mammal`, that is, '`new Mammal`'. Now, when the object `a1` is casted into `Reptile` at line1, it throws runtime exception called `InvalidCastException` because a `Mammal` cannot be cast into a `Reptile`. This is known as assignment problem.

The assignment problem is caused because the object `a1` was not identified for the reference that it was holding. Since the object `a1` is polymorphic, it is possible for it to hold a reference of any of its derived type. However, a simple attempt to cast it back to a derived type raises the risk of throwing an `InvalidCastException`. To handle this problem, C# provides the '`is`' and '`as`' operators, whereas Java provides the '`instanceof`' operator. These operators help to check the type of the reference before assigning it to any object. The C# code given in Code Snippet 9 can be rewritten to avoid the `InvalidCastException` as shown in Code Snippet 10.

Code Snippet 10:

```
class Animal
{
    public virtual void Move()
    {
      Console.WriteLine("Moving");
    }
}
class Mammal : Animal
{
    public override void Move()
    {
      Console.WriteLine("Mammal Moving");
    }
}
class Reptile : Animal
{
   public override void Move()
    {
      Console.WriteLine("Reptile Moving");
    }
}
class CastSafe
{
  public void ApplyIsOperator(Animal a1)
  {
    if (a1 is Mammal)       // line1
    {
       Mammal m = (Mammal)a1;
       m.Move();
    }
```

```
    else if(a1 is Reptile)
    {
       Reptile r = (Reptile)a1;
       r.Move();
    }
  }

  public void ApplyAsOperator(object o)    // line2
  {
    Mammal m = o as Mammal;    // line3

    if (m != null)   // check if 'm' has reference of Mammal
    {
      Console.WriteLine("I am a Mammal");
    }
    else
    {
      Console.WriteLine("{0} is not a Mammal", o.GetType().Name);
    }
  }

  static void Main()
  {
    CastSafe c1 = new CastSafe();

    Animal a1 = new Reptile();

    // Using 'is' operator to verify the type before casting

    c1.ApplyIsOperator(a1);       // line4

    // Using 'as' operator to test for null reference

    c1.ApplyAsOperator(a1);     // line5
  }

}
```

```
Output:
Reptile Moving
Reptile is not a Mammal
```

Code Snippet 10 shows the use of 'is' and 'as' operators in C# to solve the identification problem.

The 'is' operator is used to check whether the type of an object supplied at runtime is compatible with the destination type or not. In other words, the 'is' operator is used to verify that the type of an object is what it is expected to be.

Similarly, one can use the 'as' operator to perform type conversions between compatible types. Actually, operator 'as' performs a similar role as 'is' but in a slightly different manner. To use operator 'as', each value is first casted into the destination type and then verified for a null reference.

In Code Snippet 10, there are two methods namely, `ApplyIsOperator()` and `ApplyAsOperator()`. The `ApplyIsOperator()` method takes an object of class `Animal` as parameter. Within the method, the operator 'is' is used to check whether the reference object `a1` is holding is of `Mammal` or `Reptile` as shown in line1. According to the comparison, the appropriate casting is performed and the method `Move()` is invoked.

The method `ApplyAsOperator()` takes an Object class as a parameter. Within the method, first the object is casted to class `Mammal`. Then, the method checks if object `m` holds a reference of `Mammal` or not. If it holds the reference then the method displays '**`I am a Mammal`**' else it displays the message '**`<Object> is not a Mammal`**'.

Inside `Main()`, an object `c1` of class `CastSafe` is created. First, the method `ApplyIsOperator()` is invoked at line4 and a reference of class `Reptile` is passed as a parameter. Within the method, the object is verified as a `Reptile` using the 'is' operator and the output is '**`Reptile Moving`**'. Next, the method `ApplyAsOperator()` is invoked at line5 and a reference of class `Mammal` is passed to it. Inside the method, the object is verified whether it is a `Mammal` and the output is '**`Reptile is not a Mammal`**'. Thus, the problem of identification and assignment resulting from reverse polymorphism are resolved.

Note: C++ does not provide any special operator to resolve the identification and assignment problems caused by Reverse polymorphism. Here, the programmer has to use logic for identification of the type of child class reference held by the parent class object.

Concepts

Summary

- A polymorphic variable is one that can hold reference of more than one type of object.
- A pseudo-variable is an invisible receiver that works inside of a method and is normally not declared like a normal variable.
- The pseudo-variable is called 'this' in C++, C# and Java, 'self' in Small talk, and 'current' in Eiffel.
- The pseudo-variable eliminates the need to give different names to the method parameters from that of the actual parameters to resolve name ambiguity.
- Reverse polymorphism is a process of undoing a polymorphic assignment.
- The problems of identification and assignment can be resolved in C# by using the 'is' and 'as' operators, whereas in Java by using the 'instanceof' operator.

Check Your Progress

1. _____ is a variable that is declared as one type but holds a value of a different type.

A)	Constant variable
B)	Polymorphic variable
C)	Local variable
D)	Global variable

2. Match the following code snippet with their corresponding polymorphic type.

Code Snippet		Polymorphic Type	
A)	`Animal a1 = new Mammal();` `Mammal m = (Mammal) a1;`	**1)**	Simple polymorphic variable
B)	`Animal a1 = new Mammal();`	**2)**	Pseudo-variable

C)	`string name;`
	`Animal(string name)`
	`{`
	`  this.name = name;`
	`}`
		3)	Reverse polymorphism

3. Which of the following options about polymorphic variable are true and which ones are false?

A)	A polymorphic variable is one that can hold reference of more than one type of object.
B)	Polymorphic variable assignment can be undone using Reverse polymorphism.
C)	It is not possible to explicitly use a polymorphic pseudo-variable.
D)	A polymorphic variable can hold reference of a class other than its child class.

4. John is a developer in MicroTech Corporation. He has been assigned a task to prepare a class Animal in which he has to accept the animal type from the user in a method and assign it to the instance variable named 'aType' without using different names for the instance and formal variables. John writes the following code to accomplish this task:

Check Your Progress

```
class Animal
{
  private string aType;

  public void setAType(string aType)
 {
      aType = aType;
  }

  static void Main()
  {
       Animal a1 = new Animal();
       a1.setAType("Mammal");
   }
 }
```

The code written by John failed to compile and did not give the desired result. What correction should John make in his code to run it successfully?

A)	John should use a constructor to initialize the value of variable 'aType'.

B)	John should make the variable 'aType' public and directly assign it a value.
	`public string aType;`
	`static void Main()`
	`{`
	`Animal a1 = new Animal();`
	`a1.aType="Mammal";`
	`}`

C)	John should change the name of the method parameter to 'aType1'. `aType = aType1;`
D)	John should use the pseudo-variable 'this' with the instance variable 'aType'. `this.aType = aType;`

Check Your Progress

4. Which of the following is the correct syntax for creating a polymorphic variable?

(**Assumption**: Parent is the base class and Child is the derived class)

A)	`Child c1 = new Parent();`
B)	`Parent p1 = new Parent();`
C)	`Parent p1 = new Child();`
D)	`Parent = Child;`

6. Match the following pseudo-variable names with their corresponding languages.

	Pseudo-variable		Language
A)	this	**1)**	Small Talk
B)	current	**2)**	C++, C#, and Java
C)	self	**3)**	Eiffel

7. Consider the following code:

```
class Ball
{
    public void move() {}
}
class BlackBall : Ball
{
   public void move()
   {
      Console.WriteLine("Black ball moving");
    }
}

class WhiteBall : Ball
{
   public void move()
   {
```

Check Your Progress

```
    {
        Console.WriteLine("Black ball moving");
    }
}

class Box
{
    public void add(Ball b1)
    {
        BlackBall bb = (BlackBall)b1;
        bb.move();
    }

    static void Main()
    {
        Box bx = new Box();
        WhiteBall wb = new WhiteBall();
        bx.add(wb);
    }
}
```

What will be the output of the code?

A)	The code will execute successfully and print 'White ball moving' on the console.
B)	The code will execute successfully and print 'Black ball moving' on the console.
C)	The code will throw an InvalidCastException.
D)	The code will not give any output.

Session
10

Generics

Concepts

Objectives

At the end of this session, the student will be able to:

- *Explain Generics*
- *List ways of implementing Generics*
- *Explain Template Function*
- *Explain Template Class*
- *List advantages and disadvantages of Generics*

10.1 Introduction

In Object-oriented programming, while creating an object of a class, it is required to know beforehand the type of the reference and the object to which it is being assigned. This makes the code fixed and less flexible. It would be useful to have a technique by which the type of an object can be assigned at runtime without breaking the code. OOP provides such a feature called generics which allows creating such data members and functions, where the type is not specified at compile time but is applied at runtime.

This session explains the concept of Generics used to create general purpose classes and functions. Further, this session explains about different ways of implementing generics by using Template Functions and Template Classes. Finally, the session describes the advantages and disadvantages of Generics.

10.2 Generics

In general, the word 'generic' means 'general' or 'universal'. It applies to something which is generally applicable. Consider a situation where a user named Mark wishes to enter a value of his choice and store it into a variable. In this case, he would require a variable that can store any kind of value supplied by him at runtime. That is not possible because traditionally a variable has to have a type so that it can store a value. However, OOP provides a way of creating a general purpose variable or function whose type is decided by the value passed by the user at runtime using generics.

A generic function or class is one whose type is not set at compile time. It is parameterized by values of unspecified type. Generic is also known as a Template in C++. The type of the value of a generic function or variable is decided later based on value supplied by user. Figure 10.1 shows an example of generics.

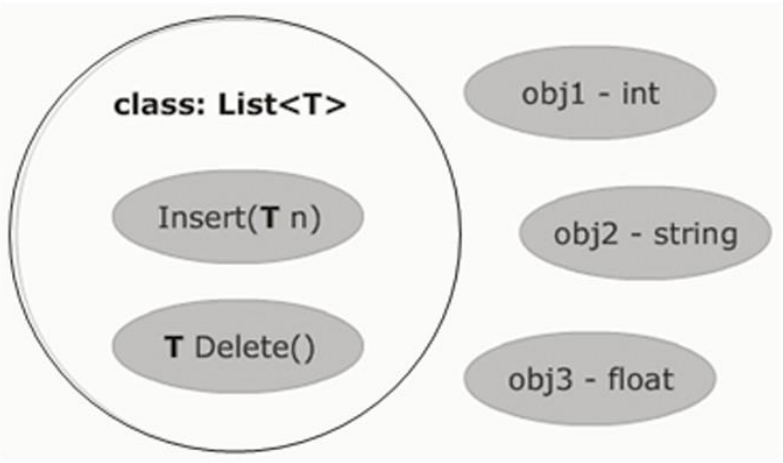


Figure 10.1: Generics

Figure 10.1 shows a class `List` with two member functions namely, `Insert()` and `Delete()`, which will be used to add or remove a value from the list. However, the type of the value is not fixed. It is indicated by the symbol 'T' which means Type. The type will be decided based on the value supplied by the user while creating the object. It could be int, float, string, or even an object type.

Generics, also known as templates in C++, are a way of making a class more abstract. It allows the programmer to define the behavior of the class without knowing the data type that will be handled by the operations of the class. This is what is called generic programming. Templates can be used together with abstract data types to allow them to handle any type of data. It allows a function or class to work with different data types without having to rewrite it for each one. For example, one could make a template List class that can handle a list of any data type, rather than creating a List class for every different data type for which one wants the list to function.

With generics, a name is defined as a type parameter. This parameter is then used inside the class as if it were a type. However, no properties of the type are known during class compilation. It is during runtime that the type parameter 'T' is matched with a specific type supplied by the user.

Generics are applied in different ways by different programming languages. C++ uses the keyword 'template' to indicate that a class is a template class using generics. However, C++, C#, and Java, all use the symbol 'T' to indicate the parameter type to be replaced at runtime.

10.2.1 Template Functions

A template function has the same syntax as that of a normal function. However, unlike a normal function, the template function does not have a specific type attached to the parameters or even the return type.

The type for the parameters is decided based on the value passed by the user at runtime.

The syntax for writing a template function is shown here:

Syntax:

```
<access-modifier> <T> <method-name> (<T> <parameter-name>,..)
{
     // processing statements
     <return-value (or expression)>
}
```

where,

`access-modifier`: States the visibility of the method to others.

`<T> method-name`: The return type and name of the method being declared.

`<T> parameter-name`: The type and name of arguments inside the method declaration.

`return-value (or expression)`: The value or expression to be returned from the method.

The entire syntax of the template function is same as a normal method except the return-type and the data type of the parameters that have been replaced by the symbol 'T'. An example of template function is shown in Code Snippet 1.

Code Snippet 1:

```
public T add(T n1, T n2)
{
    return (n1 + n2);
}
```

Code Snippet 1 shows a template function `add()` that takes two parameters `n1` and `n2`. The type of the parameters is not fixed yet. The function adds the two numbers and returns the addition. The return type is also not fixed yet. It will be set once the user passes values to the parameters `n1` and `n2`.

A template function can also be overloaded as the normal method. This is shown in Code Snippet 2.

Code Snippet 2:

```
public T add(T n1, T n2)  {}
public T add(T n1, T n2, T n3) {}
```

Concepts

10.2.2 Template Class

A template class is similar to a normal class except that it consists of a more generic type declaration. The generic type introduces a type variable named 'T' which can be used anywhere inside the class. This technique can be applied to interfaces as well. Here, the use of 'T' is not anything complex. In fact, it is quite similar to what one already knows about variables in general. You can think of 'T' as a special kind of variable whose value will be decided based on user's input. This could be any class type, interface type, or even another type variable. In fact, you can say that 'T' is a formal type parameter of a class. The syntax of writing a template class is shown here.

Syntax:

```
<access-modifier> class <class-name> <T>
{
      <data-members>

      <methods>
}
```

where,

`access-modifier`: Used to restrict access to the class.

`class-name <T>`: Name of the class with its type.

`data-members`: Attributes declared in the class.

`methods`: Functions declared in the class.

The structure of a class contains an `access-modifier` followed by the keyword `class` and `class-name`. The class name is followed by the symbol 'T' which indicates the type that will be used within the class. The C# code given in Code Snippet 3 shows an example of template class.

Code Snippet 3:

```
public class Calc<T>
{
   // data members
   // methods
}
```

Generics

To understand the concept of generics better, consider the example of class `List`. Usually when a list is created, the type of the values inside the list is fixed at compile time. Now, suppose if the user wishes to make a list of any type as and when required, the type of the values in the list cannot be fixed during compile time. In this case, the user can use generics and create template class and template functions to accomplish the task. Code Snippet 4 shows a template class `List` in C# that is used as a general purpose class for creating any type of list.

Code Snippet 4:

```
public class List<T>  // line1
{
   private T t;
   public void Insert(T n)   // line2
   {
        t = n;
        Console.WriteLine("Value inserted is "+t);
   }
   public T Delete()    // line3
   {
        return t;
   }
}
class Program
{
 static void Main()
 {
  List<int> myInt = new List<int>();   // line4
  myInt.Insert(10);    // line5
  myInt.Insert(20);
  int number = myInt.Delete();   // line6
  Console.WriteLine("Value deleted is "+number);
 }
}
```

Output:

```
Value inserted is 10
Value inserted is 20
Value deleted is 20
```

Code Snippet 4 shows a class called `List` with two methods namely `Insert()` and `Delete()`. The symbol 'T' shown in line1, line2, and line3 is used to indicate the type of the variable which will be supplied later. While creating the object `myInt` of class `List` in line4, the user supplies the value `<int>`.

This is known as generics. This tells the compiler that the variable is going to hold an `integer` value as shown in line5. Similarly, the return type of method `Delete()` becomes an `integer` which can be then stored in an integer variable as shown in line6.

This way, by using generics, you can assign `<float>`, `<string>`, or any other class type to the object `myInt` based on the requirement and accordingly a list of that type will be created. Thus, class `List` becomes a general purpose class which can be used for any type.

10.3 Advantages and Disadvantages of Generics

The use of generics is beneficial when a user wants to have a general purpose class or function for implementing functionality. However, generics also come with a set of drawbacks. Some advantages and disadvantages of generics are listed here.

Advantages:

- Generics allow a programmer to create classes with flexibility in applying the data type that is decided when the object is created and not when the class is created.
- Generics allow overloading of template functions with parameters of unspecified types.
- The main benefit of generics is late binding of the types.
- The most common use of generics is the creation of typed Collections such as List, Stack, and Array, which removes the need for casting that was previously needed when dealing with the collections.
- Templates are considered to be 'type-safe'. That is the compiler can determine whether the type associated with the template can perform all the required functions or not.

Disadvantages:

- Some compilers are known to have poor support for templates which may lead to decreased code portability.
- Some compilers lack in clear description when they detect a template error. This may increase the effort in template development.
- The compiler usually generates additional code for each template type. Therefore, indiscriminate use of templates may cause 'code bloat' and lead to large executables.
- Templates of Templates, that is, nested templates are not supported by all compilers. Also, some compilers may have a maximum nesting level.
- The use of 'less-than' and 'greater-than' signs as delimiters causes problems for tools such as text editors that analyze the source code syntactically. It is difficult, or even impossible, for such tools to determine whether the use of these symbols is as template delimiters or comparison operators.

Summary

- Generics is a technique of making a class more abstract by defining the behavior of the class without specifying the data type that will be handled by the operations of the class.
- Generics is also known as templates in C++.
- A template function is a method that does not have a specific type attached to the parameters or even the return type.
- A template class is a generic class that introduces a type variable named 'T' which can be used as a type anywhere inside the class with its data members and methods.
- Generics allow overloading of template functions with parameters of unspecified types.
- The language C++ uses the keyword 'template' to indicate that a class is a template class using generics.
- The most common use of generics is the creation of typed Collections such as List, Stack, and Array.

Check Your Progress

1. _____ is a method that does not have a specific type attached to the parameters or even the return type.

A)	Overridden method
B)	Overloaded method
C)	Template function
D)	Static method

2. Identify the correct syntax for creating a template function called 'divide'.

A)	`public divide T(n1 T, n2 T)`
	`{`
	`  return (n1/n2);`
	`}`

B)	`public divide(T n1, T n2)`
	`{`
	`  return (n1/n2);`
	`}`

C)	`public T divide(T n1, T n2)`
	`{`
	`  return T;`
	`}`

D)	`public T divide(T n1, T n2)`
	`{`
	`  return (n1/n2);`
	`}`

3. Which of the following options is not an advantage of Generics or Templates?

A)	Generics allow overloading of template functions with parameters of unspecified types.
B)	Templates are considered to be 'type-safe'.
C)	Generics enable making class more abstract by defining the behavior of the class without specifying the data type.
D)	Nested templates are not supported by all compilers.

Check Your Progress

4. Mary is a developer in TechSolutions company. She has been assigned a task to create a general purpose class called 'Stack' which can be used to create a stack of any type of value or object. What is the best technique that Mary can use to accomplish the task?

A)	Mary should use method overriding.
B)	Mary should use generics.
C)	Mary should use inheritance.
D)	Mary should use polymorphic variable.

5. John has created the following template function to add two values of any type.

```
public T add(T n1, T n2)
{}
```

Now, John wishes to add three values. What should he do to accomplish this?

A)	John should create a function with a different name and three parameters.
B)	John should delete the `add()` function with two parameters and create a new `add()` function with three parameters.
C)	John should create an overloaded template function with three parameters. `public T add(T n1, T n2, T n3) {}`
D)	John should create a derived class and change the signature of the `add()` method in the derived class.

Check Your Progress

6. Susan has written the following code to create a generic class called `Box` in which she can store any type of values. However, when she executes the code, it is giving incorrect output. Identify the line which might be causing the problem?

```
public class Box<T>
{
   private T t;
   public void Add(T n)
   {
        t = n;   // line1
        Console.WriteLine("Value inserted is "+t);
   }
static void Main()
 {
  Box<int> myBox = new Box<int>();     // line2
  myBox.Add(10);      // line3
  myBox.Add('c');     // line4
 }
}
```

A)	line1
B)	line4
C)	line3
D)	line2

Session
11 Frameworks and Reflection

Objectives

At the end of this session, the student will be able to:

- *Explain Software framework*
- *List examples of Frameworks*
- *Explain Microsoft .NET framework and Java AWT API*
- *List Advantages and Disadvantages of Framework*
- *Explain Reflection*
- *List examples of Reflection*
- *Explain Java and C# Reflection API*
- *List Advantages and Disadvantages of Reflection*

11.1 Introduction

Today, information technology is developing with leaps and bounds. Everyday new software and technologies are invented and launched into the market. However, while creating a new technology or upgrading an existing technology, it is very difficult and time consuming to start creating the software from scratch. There are technologies and software in which existing components can be used again. For example, Login page is a common entity in almost all Web sites. Therefore, creating the Login page again and again for each Web site would be unnecessary and time consuming. Similarly, the controls on a Web page such as buttons need not be recreated again. In such cases, it would be useful to have a way of storing the related components as a unit and reusing them later. Object-oriented programming (OOP) provides such a facility in the form of software Frameworks.

Now, while creating software, you may require the code to introspect itself. That is, you may want a way to examine the classes, methods, and such other parts of the code to manipulate the internal properties of the code components. For this purpose, OOP provides the concept of Reflection.

This session explains the concept of Frameworks and Reflection. Further, this session explains about the different types of frameworks with advantages and disadvantages. Finally, the session describes reflection with its advantages and disadvantages.

Concepts

11.2 Frameworks

In general terms, a 'framework' can be considered as a basic structure behind a system, concept, or text. You must have noticed that when a building is under construction, first the entire building's basic structure is prepared. Once the base is ready, the cementing, plastering, and coloring starts as per user's requirement. Any changes that are made in the building as per user's requirement are only superficial but the basic structure is not changed. Figure 11.1 shows a building with its basic framework.

Figure 11.1: Framework of a Building

Figure 11.1 shows the framework of a building under construction. Once the framework is ready, the designing of the rooms and other parts of the building can be started as per user's requirement. This is just the basic construction model which can be applied to any other building when required.

Similarly, in OOP, a framework is a set of reusable software components or classes that form the basis of an application. Earlier, it was very difficult to develop complex applications. Now, it is very easy to create such applications due to availability of a variety of frameworks.

The key idea behind development of frameworks is that in OOP, inheritance can be used in two different ways. First, it can be used as a technique for code reuse allowing code abstractions to be carried from one project to another. Second, it can be used as a technique to implement specialization. That is, it allows using a general purpose tool for doing a more specific task by extending its functionality. To understand framework better, one can divide the methods in a class into two broad categories:

1. **Foundation methods** – These are methods that get inherited from the parent and reflect the reuse of code supplied by the parent. They are defined in a parent class and become part of a child class through inheritance but are not overridden.

2. **Specialization methods** – These methods are declared in the parent but implemented in the child and thus serve as an example of reuse of concept. Such methods are also called deferred methods. These methods change the behavior of the parent class to fit the requirement of child class, that is, they are overridden in the child.

Frameworks consist of sections known as frozen sections and hot sections. Frozen sections define the overall architecture of a system such as its basic components and the relationships between them. These sections remain unchanged or frozen in any application of the framework. Hot sections represent those parts where the developers using the framework add their own code to achieve the functionality specific to their own project.

In OOP, a framework can be created for the layers of an operating system, for a group of functions in a system, the layers in a network, or layers of an application subsystem, and so on. Thus, you can say that framework is a reusable library of classes and functionality. It helps you to design applications using predefined and consistent designs and patterns.

11.3 Example Frameworks

OOP languages offer a variety of frameworks. Each of these frameworks is useful in creating some type of application. For example, Microsoft provides the .NET Framework and Java provides frameworks such as AWT, Struts, Hibernate, JSF, Spring, and other such frameworks for creating complex Web and Enterprise applications.

The most widely used application frameworks are those used for creation of Graphical User Interfaces (GUI). However, the concept can be applied beyond development of GUI. For example, there exist frameworks that are designed to build editors for various domains, for compiler construction, and for financial modeling applications.

11.3.1 Microsoft .NET Framework

Microsoft provides the .NET framework for creating Windows-based as well as Web-based applications. The .NET framework stack is shown in figure 11.2.

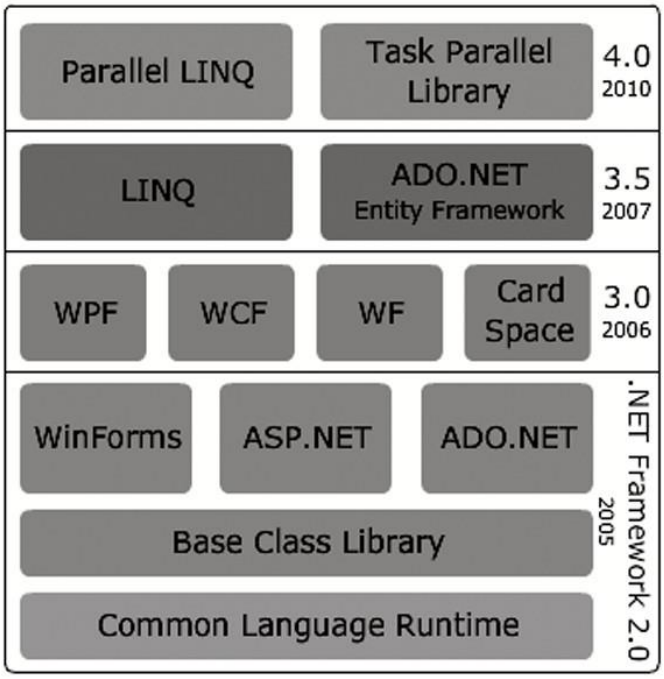


Figure 11.2: .NET Framework Stack

The .NET Framework is a software framework that is primarily designed for Microsoft Windows operating system. It consists of a large library of classes and provides language interoperability between a variety of languages such as C#, C++, VB, and other such languages supported by the framework. This means that code written in one language can be used in other languages. Programs that are based on the .NET Framework execute in a software environment rather than a hardware environment. This software environment is known as the Common Language Runtime (CLR).

CLR is a component that provides important runtime services such as security, memory management, and exception handling to the code. The CLR and the class library together form the .NET Framework.

The Base Class Library provides a set of classes for creating the user interface, Web application development, database connectivity, network communication, cryptography, and numeric algorithms. Programmers can develop software by combining .NET Framework library and other libraries with their own source code.

The .NET Framework is designed to be used for creating new and sophisticated applications for the Windows platform. Microsoft also provides a popular Integrated Development Environment (IDE) called Visual Studio for creating .NET based software. Different types of applications can be built on the .NET platform such as standalone or Windows based applications, Web-based applications, and console applications.

Session 11

Frameworks and Reflection

The .NET framework provides WinForms for creating the GUI of the .NET application. ASP.NET is used to create Web applications and ADO.NET is used to create database connectivity in .NET based applications.

Microsoft started developing the .NET Framework in the late 1990s. It was originally created under the name of Next Generation Windows Services (NGWS). The first beta version of .NET 1.0 was released by end of 2000. Since then, the .NET Framework has undergone major changes and several versions have been released. Table 11.1 shows the .NET framework versions and their compatible Visual Studio versions released so far.

.NET Version	Release Year	Visual Studio Version	Windows Default
1.0	2002	Visual Studio.NET	Windows XP Tablet and Media Center Editions
1.1	2003	Visual Studio 2003	Windows Server 2003
2.0	2005	Visual Studio 2005	Windows Server 2003 R2
3.0	2006		Windows Vista, Windows Server 2008
3.5	2007	Visual Studio 2008	Windows 7, Windows Server 2008 R2
4.0	2010	Visual Studio 2010	Windows 7
4.5 (consumer preview)	2012	Visual Studio '11'	Windows 8, Windows Server 8

Table 11.1: Microsoft .NET Framework Versions

However, Windows XP, including service packs do not come with any pre-installed version of the .NET Framework. The .NET Framework also contains two versions for mobile or embedded devices. A truncated version of the framework, called the .NET Compact Framework, is loaded on Windows CE platforms. This includes the Windows Mobile devices such as Smartphone. Additionally, the .NET Micro Framework is designed for extreme resource-constrained devices.

The higher versions of .NET Framework such as 3.0, 3.5, and 4.0 consist of new libraries such as Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Workflow Foundation (WF), Language-Integrated Query (LINQ), Card Space, and other additional libraries. A brief description of these concepts is as follows:

- WPF is a graphical subsystem in .NET used for creating rich user interfaces in Windows-based applications quickly and easily. It was previously called 'Avalon'.
- WCF is an Application Programming Interface (API) in .NET that is used for developing connected and service-oriented applications. It was originally called 'Indigo'.
- WF is a Microsoft technology that provides the user with an API, an in-process workflow engine, and a designer that can be hosted to implement long-running and complex processes as workflows within .NET applications. A workflow here is a series of well-defined programming steps or phases. In WF, each of these steps is modeled as an Activity.

Concepts

- LINQ (pronounced as 'link') is a .NET Framework component that provides native data querying capabilities to .NET languages. It provides an easy way to transfer and filter data to and from different types of data sources such as arrays, XML, relational databases, and other third party data sources.

- CardSpace is an instance of identity client software called an Identity Selector. It is used to store references to a user's digital identity and present them to the user as visual Information Cards. CardSpace was initially called 'InfoCard'. It provides a consistent GUI designed to help users to easily and securely use these identities in applications and Web sites where they are accepted. CardSpace also allows users to create their own personal or self-issued information Cards. These cards can contain any of the 14 fields of identity information such as full name, address, and such other fields. Other transactions may require a managed InfoCard. These cards are issued by a third party identity provider that makes the claims on the user's behalf, such as a bank, an employer, or any government agency.

11.3.2 Java AWT API

AWT is short form for Abstract Window Toolkit. The Java AWT is used for creating GUIs for Java programs. The class hierarchy of AWT is shown in figure 11.3.

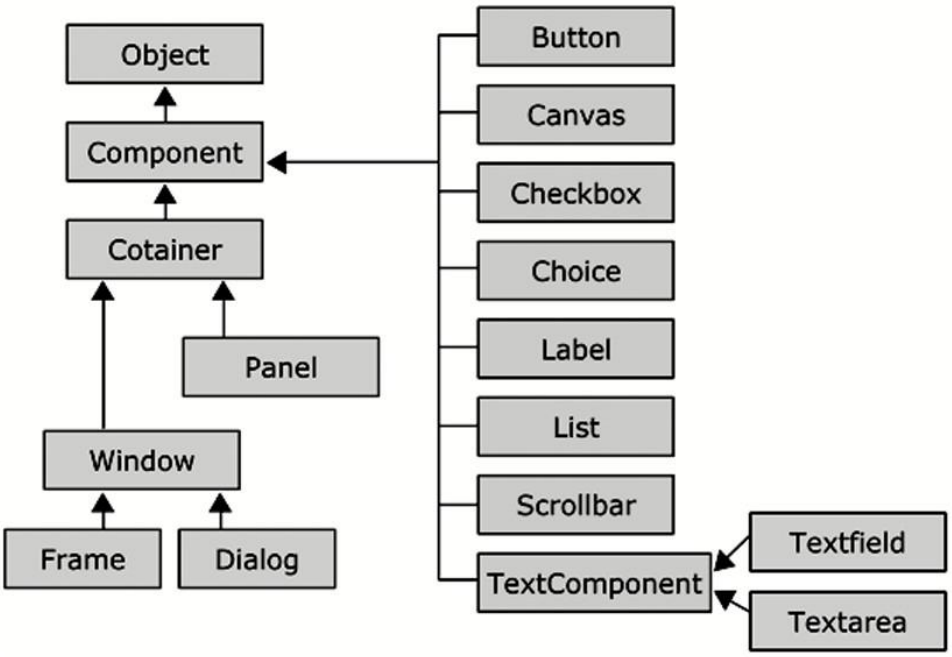


Figure 11.3: AWT Class Hierarchy

Session 11

Frameworks and Reflection

The programming language Java provides the class library called AWT. AWT is a collection of a number of common graphical widgets or components such as Button, Checkbox, List, and such other components used for designing the GUI of a Java application.

The class `Object` is the parent class of all other classes. It provides methods to check for equality of two objects, determine the class of an object, and to return a string representation of an object.

A component is any object that can be displayed on the screen with which the user can interact. A component is identified by its attributes such as size, color, location, visibility, and events associated with it. Each component displayed on the screen inherits the `Component` class. In figure 11.3, components such as `Container`, `Button`, `List`, and others are seen to inherit the class `Component`.

A Container is a special type of component that can hold other components within it. In figure 11.3, classes `Window` and `Panel` are of type `Container` and can hold other components. In a Java program, a `Window` is implemented as a `Frame` or as a `Dialog` as shown in figure 11.3.

A user can add these components or widgets to the display area and position them with a `LayoutManager`. A `LayoutManager` is an interface that designs the display area of an application based on the type of layout selected by the user. The hierarchy of the `LayoutManager` is shown in figure 11.4.

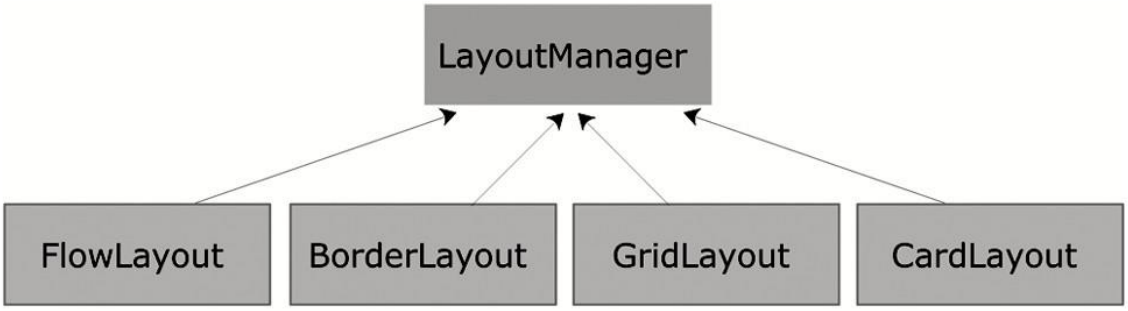


Figure 11.4: Layout Manager

Figure 11.4 shows different types of layout manager classes such as `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout`. The user can select any one or a combination of the layout managers to place the components. Each of the layouts has its own predefined design according to which it will arrange the controls when the user adds them.

AWT provides another set of interfaces called listeners to handle the actions done by users on these components. The listeners are used to handle the events generated by user actions on the components. For example, when user clicks a button, an action event is generated that is handled by the `ActionListener`. There are listeners available the various events generated by these components. The listener hierarchy is shown in figure 11.5.

Concepts

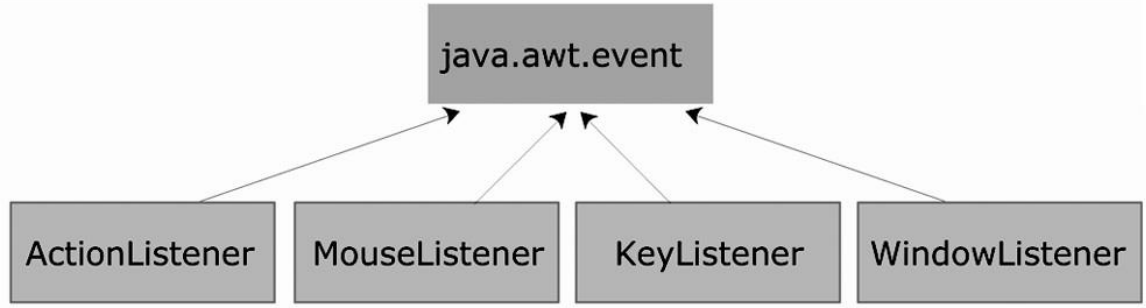


Figure 11.5: Event Listeners

Figure 11.5 shows the various event listener interfaces. The names of the interfaces are according to the component for which they handle the event. For example, `MouseListener` is used to handle events generated on the mouse, `KeyListener` for keyboard events, and so on. The listener interfaces are available in the '`java.awt.event`' package.

11.4 Advantages and Disadvantages of Frameworks

Each technology has its own benefits and drawbacks. Some advantages and disadvantages of frameworks are as follows:

➢ **Advantages:**

- A framework allows reuse of code that has been pre-compiled and tested.
- It increases the reliability of the new application created using the predefined classes and components.
- It reduces the programming and testing effort so that the developer can focus on the software development rather than on creating the tools and environment of application development.
- It helps to establish standard programming practices and correct use of design patterns and programming tools.
- An upgrade to the framework can provide new functionality and improved performance without additional programming effort for the framework user.

➢ **Disadvantages:**

- For a beginner, it is difficult to use the framework initially due to its large and complex structure that requires some time to learn.
- It is not always possible to create software simply from the resources available in the framework. For specific applications, it is required to extend the framework as per requirement.
- Creating a framework can be difficult and time-consuming.
- Over a period of time, due to repeated upgrades, a framework tends to become increasingly complex.

11.5 Reflection

In general, 'reflection' means a mirror image of an object. In daily routine, people use a mirror to see their reflection in it for examining themselves. The reflection in the mirror shows the details about the person or object standing in front of it. This is shown in figure 11.6.

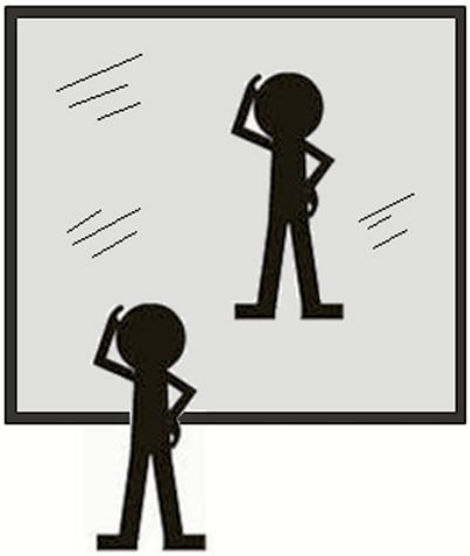


Figure 11.6: Reflection

Figure 11.6 shows the mirror image or reflection of a person in the mirror. To 'reflect upon', also means 'to give a careful consideration or thought' to something. Similarly, in OOP, reflection is the ability of a program to 'learn' something about itself. It allows a program to 'examine' or 'introspect' itself.

Concepts

Reflection enables the code to retrieve information regarding the fields, methods, constructors, and other members of the class and use that information to manipulate the internal properties of the program. Therefore, you can say that reflection is a process by which a program can retrieve its own metadata that is data about data. Thus, the program is said to reflect upon itself. The program can use this metadata either to give some information to the user or to modify its own behavior. For example, using reflection, a class can retrieve and display the names and other properties of all its members to the user.

11.6 Examples of Reflection

The ability of a program to introspect itself may not sound great but there are programming languages in which this feature does not even exist. For example, programs written in languages such as Pascal, C, or C++ do not have any way of obtaining information about the functions defined within the program. Languages such as C# and Java consist of the feature of reflection.

11.6.1 C# Reflection API

Reflection is the ability of a code to retrieve its own metadata for the purpose of finding modules, assemblies, and type information at runtime. In C#, reflection allows a program to find out details of an object, method, as well as create objects and invoke methods at runtime. C# provides the `System.Reflection` namespace that contains the classes and interfaces that can be used to get information about types, fields, and methods of the code. In C#, while using reflection, the user can use the `GetType()` method to retrieve the type of the current instance or the `typeof` operator to get the type of any object. The C# code given in Code Snippet 1 shows an example of reflection.

Code Snippet 1:

```
using System;
using System.Reflection;
public class Calc
{
  public int Add(int num1,int num2)
  {
    return (num1 + num2);
  }
 static void Main()
 {
  Console.WriteLine("Reflection");
  // Create an object of class Calc
  Calc c1 = new Calc();
  // Get Type information.
  Type myType = c1.GetType();   // line1
  // Get Method Information.
```

```
    MethodInfo m_info = myType.GetMethod("Add");     // line2

  Console.WriteLine("Class name is " + myType.FullName); // line3
  Console.WriteLine("Class is abstract - "+ myType.IsAbstract);
  Console.WriteLine("Method name is " + m_info.Name);  // line4
  Console.WriteLine("Method is a constructor - " + m_info.IsConstructor);
  }
}
```

Output:

```
Reflection
Class name is Calc
Class is abstract - False
Method name is Add
Method is a constructor - False
```

Code Snippet 1 shows the use of the reflection API. It can be included in the program by writing '`using System.Reflection`' at the beginning of the program. The code consists of a class `Calc` with one method named `Add()`. This method takes two integer parameters and returns the addition as an integer.

In the `Main()` method, an object `c1` of class `Calc` is created. Next, at line1, the `Type` class is used to get the information of the class using the `GetType()` method. The `Type` class is at the root of the reflection classes. It contains a representation of the `type` of an object. The `Type` class is the main or primary way to access metadata. Now, the object `myType` will have the required information about class `Calc`. Therefore, using the object `myType`, you can check if the class is an abstract class or a concrete class by using any of statements as follows:

`myType.IsAbstract` – to check for abstract class

`myType.IsClass` – to check for concrete class

Similarly, to get the `Add()` method's information, an object `m_info` of `MethodInfo` class is created at line2. The name '`Add`' is passed to the method `GetMethod()`. Now, the object `m_info` holds the information about the method `Add()`. Using `m_info`, you can check if the method is a constructor or a static method and many other such properties as follows:

`m_info.IsConstructor` – to check if method is a constructor

`m_info.IsStatic` – to check if method is static

In Code Snippet 1, the fully qualified name of the class is retrieved using the `Name` property of the `myType` object of `Type` class at line3. Similarly, at line4, name of the method is retrieved using the `Name` property of the `m_info` object of `MethodInfo` class.

Concepts

There are many such classes and methods in the reflection API that help a programmer to examine the code or display useful information to the user. The Java code given in Code Snippet 2 shows an example of reflection.

Code Snippet 2:

```
import java.lang.reflect.*;
public class Calc
{
  public int Add(int n1, int n2)
  {
      return (n1+n2);
  }

  public static void main(String args[])
  {
   Class c1=null;
   try
   {
     c1 = Class.forName("Calc");  // line1

     // array of Method objects
     Method mlist[] = c1.getDeclaredMethods();  // line2
     for(int i=0; i<mlist.length; i++)
     {
       Method m1 = mlist[i];
       System.out.println("Method name = " + m1.getName());
       System.out.println("Class = "+ m1.getDeclaringClass());
       System.out.println("Return type = " + m1.getReturnType();
       System.out.println("-----");
     }
   }
   catch(Exception ex)
   {
       System.out.println(ex.getMessage());
   }
  } // end of main
} // end of class
```

```
Output:
Method name = Add
Class = class Calc
Return type = int
-----
Method name = main
Class = class Calc
Return type = void
-----
```

Code Snippet 2 shows the use of the Reflection API in Java. It can be included in the program by writing '`import java.lang.reflect.*`' at the beginning of the program. The code consists of a class `Calc` with one method named `Add()`. This method takes two integer parameters and returns the addition as an integer.

In the `main()` method, first the class description for class `Calc` is retrieved in the object `c1` of class '`Class`' using the `forName()` method at line1. Next, at line2, `c1` is used to make a call to the `getDeclaredMethods()` to retrieve a list of methods of class `Calc` into the `mlist[]` object array of class Method. This array will contain objects holding information of each method defined in the class. If `getMethods()` is used in the program instead of `getDeclaredMethods()`, one can obtain information about inherited methods also.

Once a list of `Method` class objects is retrieved, a for- loop is used to display the information on the name, parameters, class, and other such properties of the method. In Code Snippet 2, the name, declaring class, and the return type of the methods `Add()` and `main()` are displayed using the `getName()`, `getDeclaringClass()`, and `getReturnType()` methods of the object `m1` of `Method` class. Similarly, other built-in classes and interfaces of the `java.lang.reflect` package can be used to retrieve information about the program constructs.

11.7 Advantages and Disadvantages of Reflection

Reflection is a very powerful tool, but it is advisable to not use it indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. Some advantages and disadvantages of reflection are as follows:

- **Advantages:**
  - Reflection allows a developer to retrieve metadata about the classes, methods, and other programming constructs in the code.
  - Reflection helps to performing type discovery which can be used for creating custom scripts.

- Reflection allows late binding of methods and properties based on type discovery. This facilitates dynamic invocation of methods.

➢ **Disadvantages:**

- Reflection induces performance overhead since it involves types that are dynamically resolved. Thus, reflective operations have slower performance than the non-reflective ones, and therefore should be avoided in sections of code that are called frequently in performance-sensitive applications.

- Reflection requires runtime permission which may not be available while running under a security manager. This is an important consideration for a code that has to run in a restricted security context.

- Reflection allows code to perform operations such as accessing private fields and methods that would be considered illegal in a non-reflective code. This may result in unexpected side-effects that may render the code dysfunctional and destroy portability.

Summary

- A framework is a set of reusable software components or classes that form the basis of an application.
- Foundation methods are one that become a part of a child class through inheritance but are not overridden in the child class.
- Specialization methods are declared in the parent but implemented in the child, that is, they are overridden in the child class.
- .NET Framework is a software framework that consists of a large library of classes and provides language interoperability between the various languages supported by the framework.
- Java provides the class library called Abstract Window Toolkit (AWT) which is a collection of a number of common graphical components used for designing the GUI of a Java application.
- Reflection is the ability of a program to 'examine' or 'introspect' itself.
- C# provides the 'System.Reflection' namespace and Java provides the 'java.lang.reflect' package for implementing reflection in the program.

Check Your Progress

1. _____ is a set of reusable software components or classes that form the basis of an application.

A)	Pattern
B)	Framework
C)	Package
D)	Assembly

2. Match the following .NET framework features with their corresponding description.

.NET Feature		Description	
A)	WPF	**1)**	Used to implement long-running and complex processes as workflows
B)	WF	**2)**	Used for developing connected and service-oriented applications
C)	LINQ	**3)**	A graphical subsystem used for creating rich user interfaces
D)	WCF	**4)**	Provides native data querying capabilities to .NET languages

3. Mark is a developer in MicroTech Corporation. He has been assigned a task to create a list of all methods of the classes of an application. Which feature of OOP should Mark used to accomplish this task?

A)	Overloading
B)	Inheritance
C)	Framework
D)	Reflection

4. Which of the following options is not an advantage of reflection?

A)	Reflection allows a developer to retrieve metadata about the program.
B)	Reflection helps to performing type discovery.
C)	Reflection allows late binding of methods and properties based on type discovery.
D)	Reflection leads to slower performance of programs.

Check Your Progress

5. Which of the following is the correct statement for including reflection feature in a C# program?

A)	`import System.Reflection;`
B)	`#include System.Reflection;`
C)	`using System.Reflection;`
D)	`set System.Reflection;`

6. Which package is used by Java to implement reflection?

A)	`java.util.reflect.*`
B)	`java.lang.reflect.*`
C)	`java.awt.reflect.*`
D)	`java.io.reflect.*`

Session
12 Patterns

Objectives

At the end of this session, the student will be able to:

- *Explain Design Patterns*
- *List the types of Design Patterns*
- *Explain the Factory and Singleton Patterns*
- *Explain the Composite and Decorator Patterns*
- *Explain the Proxy and Facade Patterns*

12.1 Introduction

In day to day life, whenever a problem is encountered, most people tend to refer to a similar problem faced in the past. The problem had been successfully solved using some technique. That technique can be used as a model and can be tried out on the new problem by making relevant changes as per the current circumstances. Similarly, in the world of programming, a previously used solution to a problem can be applied to a new problem by making relevant changes. This solution which has been used effectively in an earlier problem is described as a pattern.

This session explains the concept of design patterns. Further, this session describes the different types of patterns such as the factory, singleton, composite, decorator, proxy, and facade patterns.

12.2 Design Patterns

A pattern in real world is a shape, model, or structure appearing again and again to create a design. Take an example of the printed tiles used for flooring. The tiles are printed in such a way that they need to be arranged correctly so that the print falls in the right place and a pattern is formed. Once a pattern is formed out of the combination of first set of tiles, the rest of the flooring is completed by copying the same pattern all over. The first set of tiles became a pattern or a solution for the rest of the floor. This is shown in figure 12.1.

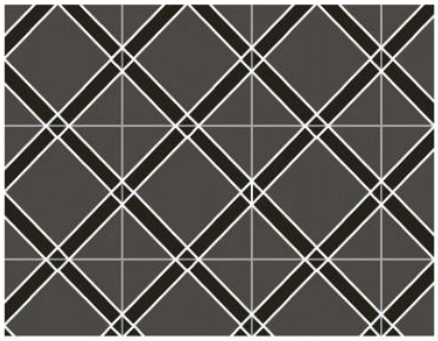


Figure 12.1: Tiles Arranged in a Pattern

Similarly, design patterns in software are strategies which are language-independent and used for solving commonly encountered object-oriented design problems. Design patterns are proven techniques for implementing robust, reusable, and extensible object-oriented software. In more technical terms, a pattern is a proven solution to a problem which has been documented so that it can be used to easily handle similar problems in the future.

Patterns have become an important concept in the development of object-oriented programming. They help in describing structures and relationships at a higher and different level of abstraction than classes, components, or instances. Documenting patterns serves two important purposes:

- It speeds up the process of finding a solution when another problem with similar characteristics is encountered. That is, it eliminates the need to 'reinvent the wheel'.
- Giving names to patterns allows programmers to use a common vocabulary to discuss design alternatives. This vocabulary is also known as pattern language.

12.3 Types of Patterns

In many different object-oriented applications, certain types of relationships appear over and over again. A pattern attempts to filter the important features of these associations. Based on the problems encountered, different patterns have been created.

12.3.1 Factory Pattern

A 'factory' in real world is a place where one or more types of products are manufactured or assembled. The products are produced based on market demand. Similarly, in OOP, the factory pattern is a pattern that refers to a factory of objects of classes. In other words, a factory class or factory method is one that is used to create different types of objects based on a user's requirement. For example, suppose you have a super class and more than one sub-classes. Now, based on the data provided by the user, you have to return the object of one of the sub-classes. In such a case, you can use a factory pattern.

Concepts

Through this pattern, you create a factory class or factory method that will create the desired object and return it to the caller. This is demonstrated in the C# code given in Code Snippet 1.

Code Snippet 1:

```
class Shape
{
  public virtual void draw()
  {}
}
class Circle : Shape
{
  public override void draw()
  {
     Console.WriteLine("Draw Circle");
  }
}
class Square : Shape
{
  public override void draw()
  {
     Console.WriteLine("Draw Square");
  }
}
class ShapeFactory
{
    public void shapeCreator(Shape s1)
    {
          if(s1 is Circle)   // line1
          {
              Circle c = (Circle)s1;
              c.draw();
          }
          else if(s1 is Square) // line2
          {
              Square s = (Square)s1;
              s.draw();
          }
    }
```

```
    static void Main()
    {
        ShapeFactory sf = new ShapeFactory();
        sf.shapeCreator(new Circle());       // line3
    }
}

Output:
Draw Circle
```

In Code Snippet 1, the classes `Circle` and `Square` inherit the class `Shape` and override the method named `draw()`. Another class named `ShapeFactory` consists of a method named `shapeCreator()` that takes an object of type `Shape` as a parameter. Within the method, the operator '`is`' is used to check if the reference object held by the argument `s1` is of class `Circle` or `Square` as shown in line1 and line2. According to the result of comparison, the appropriate casting is performed and the method `draw()` is invoked.

Inside `main()`, an object `sf` of class `ShapeFactory` is created. Next, the method `shapeCreator()` is invoked at line3 and a reference of class `Circle` is passed as a parameter. Within the method, the object is verified as a `Circle` using the '`is`' operator and the output is '**`Draw Circle`**'. Thus, the method `shapeCreator()` can be termed as a factory method, since it is responsible for creating objects of different shapes as per input given by the user.

The factory pattern can be used in the following cases:

- To create collection of dependent or related objects such as a Toolkit.
- To provide a class library that exposes interfaces without any implementation.
- To segregate concrete classes from their super classes.
- To create objects of classes at runtime based on user's input.

12.3.2 Singleton Pattern

At times, it is required to have only one instance for a class. For example, in a system, there can be only one window manager, only one file system, or one print spooler, and so forth.

The singleton pattern is one of the simplest design patterns. It involves creating only one class that is responsible for instantiating itself. This is to make sure that it creates only one instance and provides a global access point to that instance. Here, the same instance can be used from everywhere without having to directly invoke the constructor each time. Singletons are usually used for centralized management of external or internal resources. They provide a global access point to themselves.

Concepts

The implementation of singleton pattern involves creating the following members:

- a static member in the singleton class,
- a static public method that returns a reference to the static member, and
- a private constructor

Figure 12.2 shows the class diagram of a singleton class named `MySingleton`.

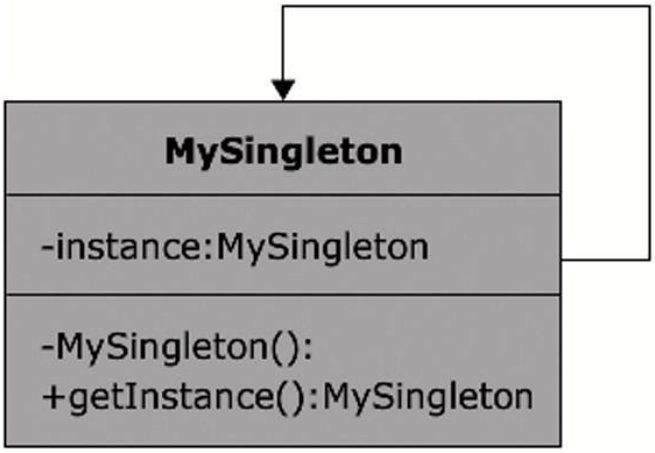


Figure 12.2: Singleton Pattern

The diagram shows a class named `MySingleton` consisting of a data member named instance which is of type `MySingleton`. The '-' symbol indicates that the member is private. The class also consists of a private constructor and a public method `getInstance()` with return type `MySingleton`. The C# code given in Code Snippet 2 shows the programmatic representation of the class `MySingleton`.

Code Snippet 2:

```
class MySingleton
{
     private static MySingleton sInstance;

      // private constructor

     private MySingleton()
```

```
    { }

    public static MySingleton getInstance()
    {
         // checking for instance existence

        if (sInstance == null)
            sInstance = new MySingleton();

        return sInstance;
    }

    public void doSomeWork()
    {
        Console.WriteLine("Only one instance is allowed");
    }
}
class Sample
{
    static void Main()
    {
        MySingleton.getInstance().doSomeWork();  // line2

    }
}
```

Output:

```
Only one instance is allowed
```

The class `MySingleton` defines a method `getInstance()` that ensures that only one instance of the class is created and returns that instance to the caller. Since the variable `sInstance` is `static`, only one copy will exist for this class. Also, since the constructor is `private`, other classes cannot directly instantiate the `MySingleton` class. The only way of instantiating the class `MySingleton` is through the `getInstance()` method. Thus, the `getInstance()` method provides a global point of access to the object of class `MySingleton` and can be used by other classes to invoke another method named `doSomeWork()` of class `MySingleton` as shown in line2.

The singleton pattern can be used when a class should have only one instance and when it must be accessible to other classes from a global access point. For example, a singleton can be used to create a common Logger class, Configuration class, and also for implementing factories as singletons.

Concepts

12.3.3 Composite Pattern

A composite pattern arranges components into a tree structure consisting of individual objects as well as composite objects. Composite objects are those that have other objects as their children. The aim of the composite pattern is to be able to treat individual and composite objects the same way. It helps to organize objects into a tree structure to depict part-whole hierarchies. Composite pattern allows a user to build a complex objects out of individual objects like a tree structure. A composite pattern consists of composite objects and individual objects also known as leaf objects. The class diagram of a composite pattern is shown in figure 12.3.

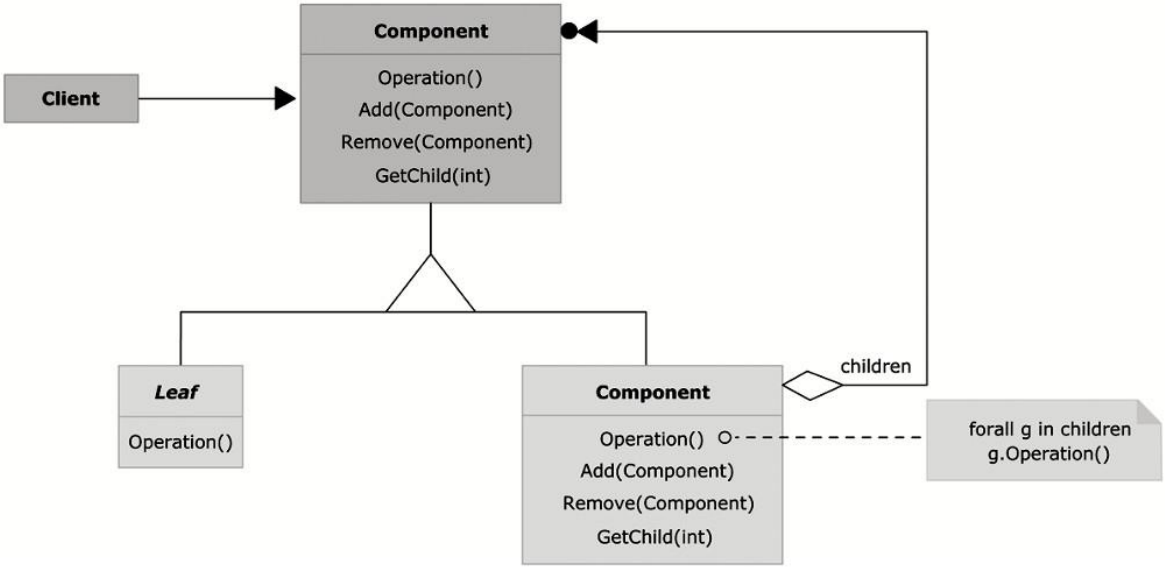


Figure 12.3: Class Diagram of Composite Pattern

The classes or interfaces participating in this pattern are as follows:

- **Component** – The root component that the client accesses.
- **Composite** – A component formed of other child components.
- **Leaf** – An individual component at the end of the hierarchy.

To understand the composite pattern, consider a GUI application that contains a form with two panels. Each panel in turn consists of other controls such as textboxes, buttons, and other such controls. This is shown in figure 12.4.

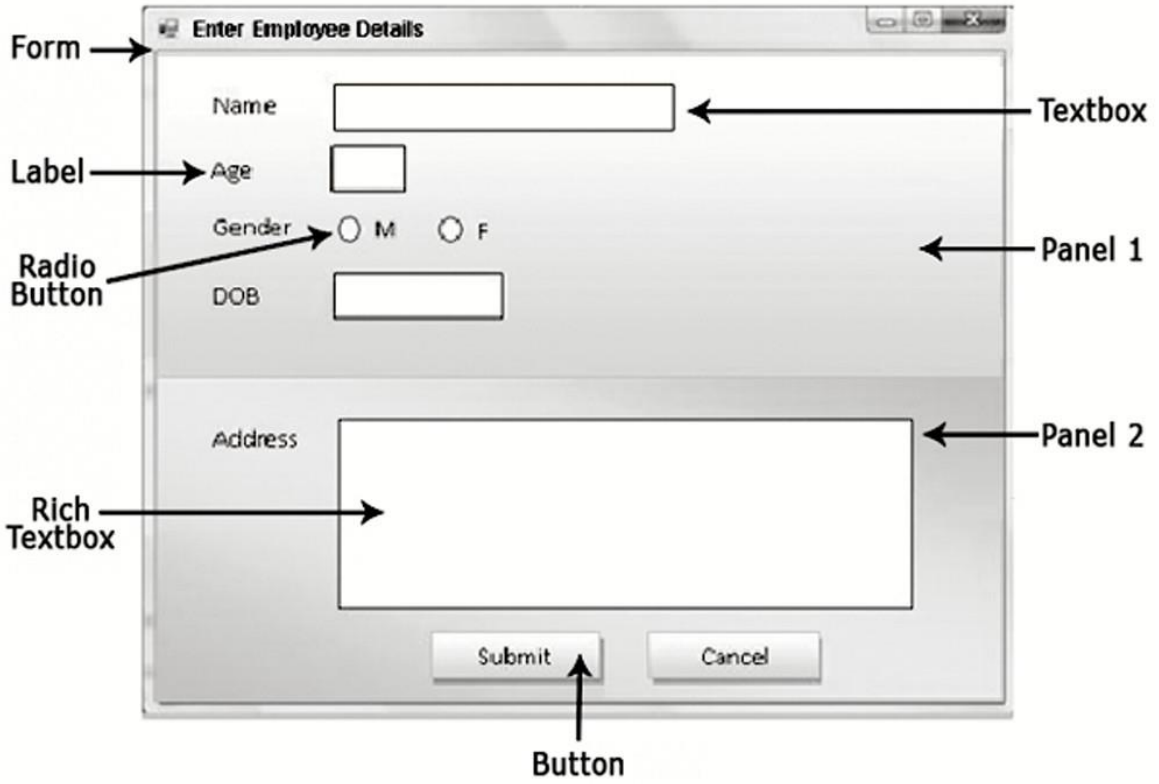


Figure 12.4: A GUI System Representing a Composite Pattern

Figure 12.4 shows a form with two panels and different controls such as textbox, button, label, radio button, and rich textbox arranged on the two panels. The tree structure of the composite pattern is shown in figure 12.5.

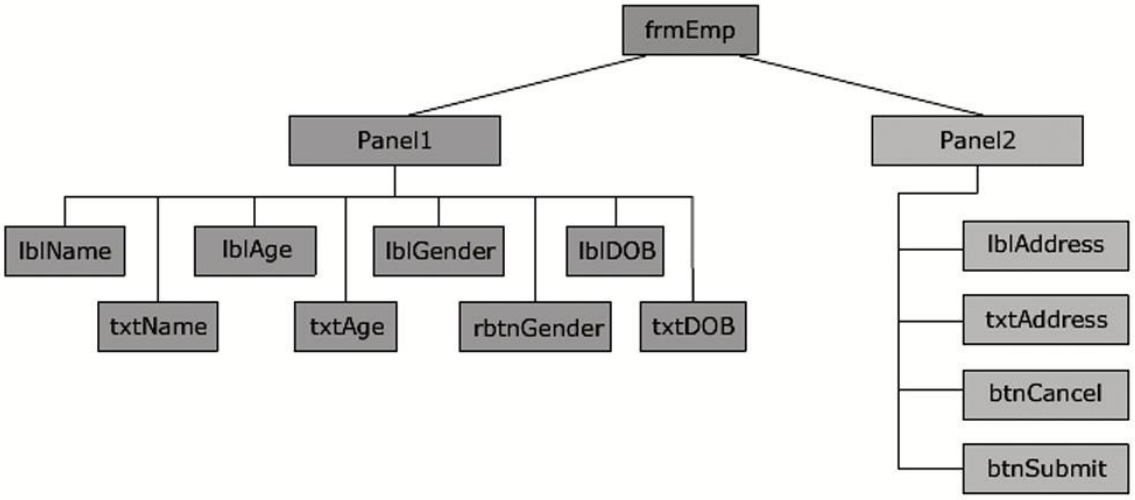


Figure 12.5: Tree Structure of the Composite Pattern

In figure 12.5, `frmEmp` is the composite root of the tree. `Panel1` and `Panel2` are intermediate composite components, whereas other controls are the leaves of the tree structure.

The composite pattern can be used in the following cases:

- When you want to represent a part-whole relationship like a tree. For example, one can represent a computer's file system using composite pattern.
- To group individual components to form larger components, which in turn can be grouped together to form still larger components.

12.3.4 Decorator Pattern

The decorator pattern is used to dynamically add or remove responsibilities to an object without affecting other objects. It provides a flexible alternative to sub-classing to extend the functionality of a class. It helps to modify or extend the behavior of an 'instance' at runtime. Usually, inheritance is used to extend the functionality of a class. However, unlike inheritance, a design pattern allows you to choose any single object of a class and modify its behavior, leaving the other instances unmodified.

To implement the decorator pattern, you need to construct a wrapper around an object by extending its behavior. The wrapper may do its job before or after you assign the call to the wrapped instance.

Figure 12.6 shows the class diagram for the decorator pattern with the participant classes.

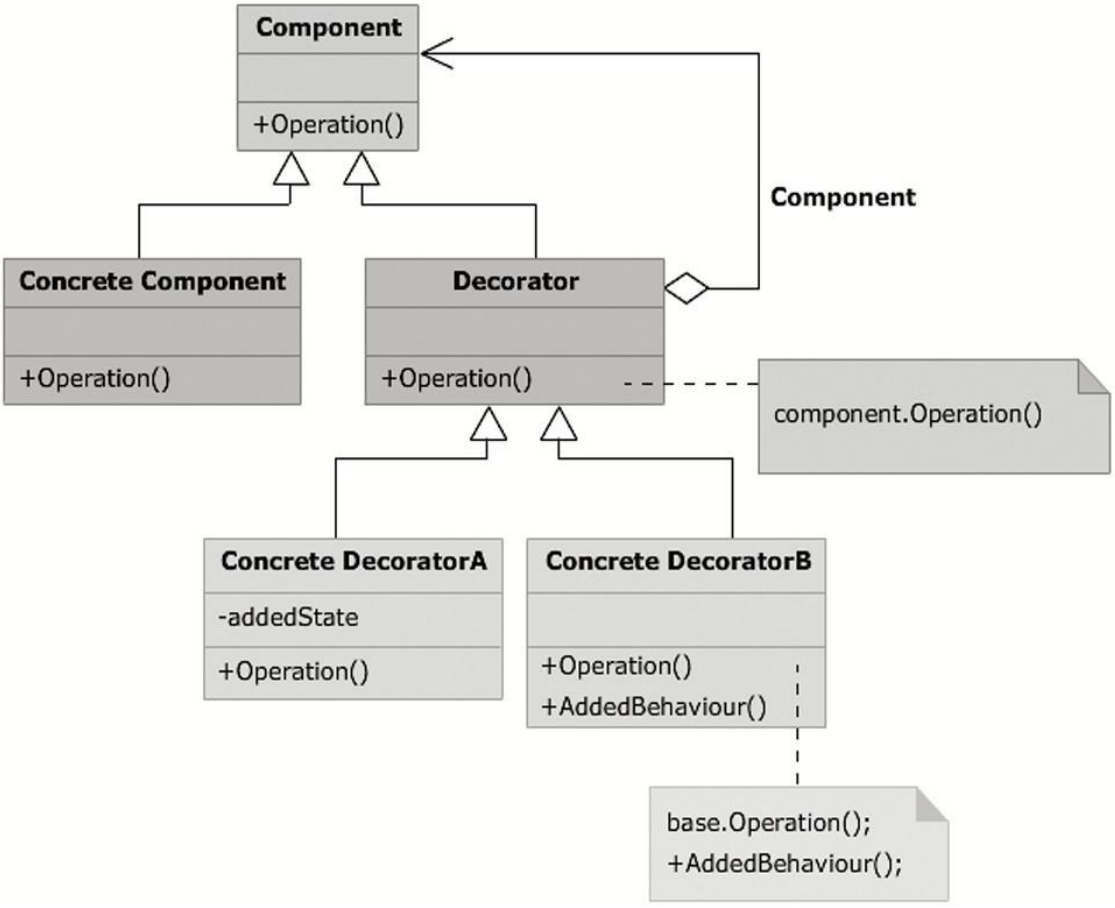


Figure 12.6: Class Diagram of Decorator Pattern

The classes or interfaces participating in this pattern are as follows:

- **Component** – It is an interface for the objects to which responsibilities can be added dynamically.
- **Concrete Component** – It defines the object to which additional responsibilities can be added.
- **Decorator** – It holds a reference to the Component object.
- **Concrete Decorators** – These classes extend the functionality of the component by adding state or behavior to it.

To create a decorator pattern, you need to create an interface. This interface creates a blueprint or method for the class which will have the decorators. Next, you need to create a class that implements that interface with its basic functionalities. Create another abstract class that contains an attribute type of the interface. The constructor of this class assigns the interface type instance to that attribute. This is the decorator base class. Now, you can extend the decorator class and create concrete decorator sub-classes. The decorator sub-classes will add their own methods. The concrete decorator sub-class will call the base class method before or after executing its own method. The key to the decorator design pattern is that the binding of the method and the base instance happens at runtime based on the object passed as a parameter to the constructor. This leads to dynamic customization of the behavior of only that specific instance.

Consider a real world analogy that illustrates this concept. Suppose you have prepared a cake. Now, you wish to decorate the cake. There are several toppings available such as icing, nuts, wafers, strawberries, chocolates, and other such items. Based on a user's selection, the cake has to be decorated.

For this purpose, one can make a decorator pattern for the cake as shown in figure 12.7.

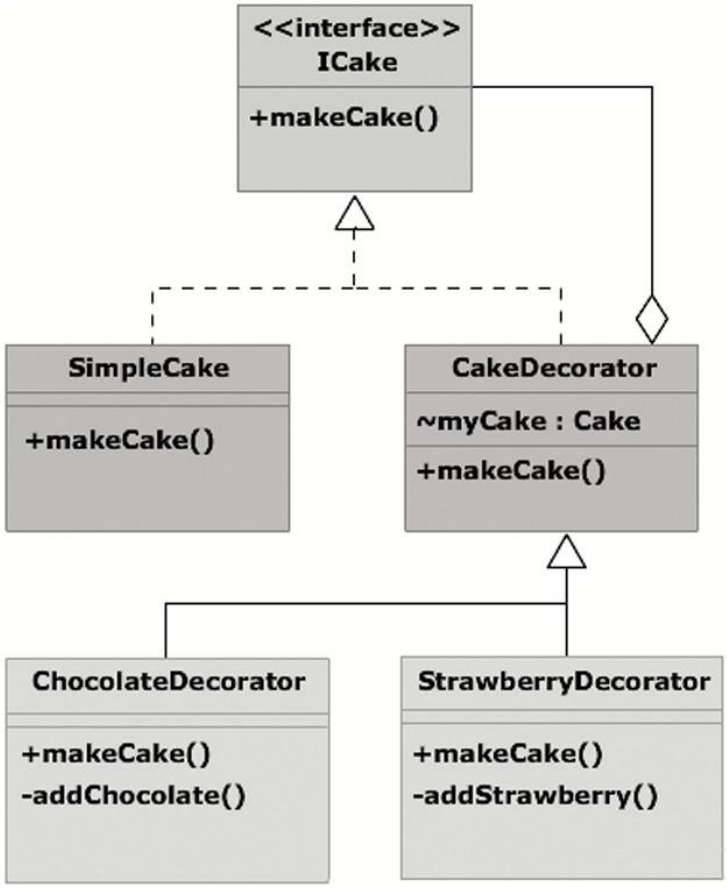


Figure 12.7: Class Diagram of Decorator Pattern for a Cake

Figure 12.7 shows an interface `ICake` depicting a cake. The class `SimpleCake` implements the `ICake` interface. This is the base class on which the decorators will be applied. Class `CakeDecorator` is the decorator class and the core of the decorator pattern. It contains a protected attribute `myCake` which is an instance of the type interface `ICake`. The instance is assigned a value dynamically at the creation of decorator through its constructor. Once assigned, the method of the appropriate instance will be invoked. The two classes `ChocolateDecorator` and `StrawberryDecorator` are the two concrete decorator classes inheriting the abstract decorator class called `CakeDecorator`. When the two classes are created, the reference of the base interface `ICake` is passed into their constructors and in turn assigned to the super class `CakeDecorator`. In the `makeCake()` method of both the decorators, first a call to the `makeCake()` method of the `ICake` interface is made, followed by their own methods `addChocolate()` and `addStrawberry()`. The `addChocolate()` and `addStrawberry()` methods extend the behavior by adding their own functionality. This is shown in the C# code given in Code Snippet 3.

Code Snippet 3:

```
// Designing the component interface

interface ICake
{
   String makeCake();
}

// Designing the concrete component that implements ICake

class SimpleCake : ICake
{
   public String makeCake()
   {
      return "Basic Cake";
   }
}

// Designing the abstract Decorator class

abstract class CakeDecorator : ICake
{
   protected ICake myCake;

        // constructor
        public CakeDecorator() { }
        public CakeDecorator(ICake myCake)
        {
            this.myCake = myCake;
        }

        public virtual String makeCake()
        {
            return myCake.makeCake();
        }
}
```

```
// Designing the Concrete Decorator sub-class1

class ChocolateDecorator : CakeDecorator
{
   public ChocolateDecorator(ICake myCake):base(myCake)
   {}

    public override String makeCake()
    {
          return myCake.makeCake() + addChocolate();
     }

    private String addChocolate()
    {
        return "+ Chocolate chips";
    }
}

// Designing the Concrete Decorator sub-class2

class StrawberryDecorator :CakeDecorator
{
   public StrawberryDecorator(ICake myCake): base(myCake)
   {}

    public override String makeCake()
    {
          return myCake.makeCake() + addStrawberry();
     }

    private String addStrawberry()
    {
        return "+ Strawberries";
    }
}
```

Concepts

```
// Executing the decorator pattern

class TestCakeDecorator
{
    static void Main()
    {
          ICake cake1 = new ChocolateDecorator(new StrawberryDecorator(new
SimpleCake())); // line 1
            Console.WriteLine(cake1.makeCake());
     }
}
```

Output:

```
Basic Cake + Strawberries + Chocolate chips
```

In Code Snippet 3, the `TestCakeDecorator` class is used to execute the decorator pattern. A reference `cake1` of the `ICake` interface is created. The object of `SimpleCake` class is then wrapped into `StrawberryDecorator` which in turn is wrapped into `ChocolateDecorator` and then assigned to `cake1`. Finally, the `makeCake()` method of `cake1` object is invoked as shown in line1. You can modify the call on line1 to suit your requirements. For example, to have only a basic cake with strawberry toppings, you can write the following code:

```
ICake cake1 = new StrawberryDecorator(new SimpleCake());
```

The decorator pattern can be used in the following cases:

- To provide an alternative to sub-classing.
- To add new functionality to an object without affecting other objects.
- To simplify the task of adding and removing a responsibility from an object dynamically.

The decorator pattern provides functionality to objects in a more flexible way as compared to inheriting from them. However, the disadvantage of the decorator pattern is that code maintenance can become difficult as it provides the system with a lot of similar looking small objects of each of the decorators.

12.3.5 Proxy Pattern

The proxy pattern uses a simple object to depict a complex one or in other words, provides a 'placeholder' for an object in order to control access to it. If it is expensive to create an object, its creation can be postponed till the actual need arises. Until then, a simple object can represent it. This simple object is referred as a 'proxy' for the complex object.

For example, suppose you wish to withdraw money from a bank. You can do it either by going to the bank and use a cheque book to withdraw money or you can use the ATM. In earlier days, when there was no ATM, people had to go to the bank to withdraw money. They had to carry their passbooks or cheque books, and wait in a long queue after which their transaction was completed. However, today people finish the same task in just a few minutes by visiting the ATM. Thus, the complex task of withdrawing money by going to the bank is simplified with the use of an ATM. Here, the ATM becomes a proxy to a bank which simplifies the money withdrawal task.

Similarly, suppose a person is developing a Web-based application in which part of the application will be hosted on one computer and other part will run on another computer linked by a network connection. Creating a direct connection between these computers and transmitting data over this connection are details irrelevant to the actual working of the application. One way to address this issue is to use a proxy. The proxy will be a link which will hide the network connection details. Objects will interact with the proxy and be least concerned about any type of network connection involved in the process. On receiving a request, the proxy will package it and transmit it over the network to the other computer. The response of the other computer will also be received by the proxy and passed back to the client. In this way, the client is completely unaware of the details of the network. Figure 12.8 shows the situation of a client making request through a proxy.

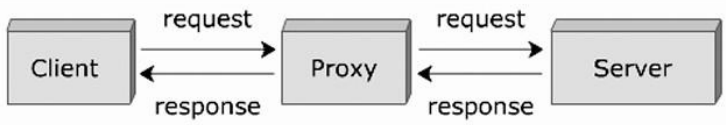


Figure 12.8: General Concept of Proxy

Figure 12.8 shows how a proxy hides the network and protocol detail from the client. Due to this pattern, the client is only concerned with sending the request and receiving the response. The client is not concerned as to how the request will be sent on the network. Similarly, in design patterns, a proxy pattern will act as a representation of an actual object. For example, you can define a proxy class to access a remote method.

The proxy pattern can be used in the following cases:

- When creating an object is too expensive in terms of time or memory.
- To postpone creation of an object until it is actually needed.
- When loading a large image which can be time consuming.
- To load a remote object over a network during peak hours.
- When access permission is required to operate a complex system.

12.3.6 Facade Pattern

In general, a 'facade' means a face or a mask that hides the actual object behind it. Similarly, the facade pattern also uses an additional layer or class like a mask to make a complex system simpler. It provides a general or unified interface, which is at a layer higher level than the subsystems.

For example, consider a Store as a complex system. This store is run by a Storekeeper. In the storage, there are several things stored such as raw materials, packing materials, and finished goods. You, as a client, want to access some materials. However, you do not know where the different goods are stored. You only have access to storekeeper who knows the details of his/her store. To obtain specific items in the store, you need to ask the storekeeper and he/she will take it out of store for you. Here, the storekeeper acts as the facade because he/she hides the complexities of the Store.

Similarly, while using a computer system, you interact with a facade, that is, the 'computer' to access the more complex internal system consisting of parts, such as CPU, memory, and hard drive. To start these devices, all you have to do is to start the computer and it internally starts all these devices also. A computer system with its internals is shown in figure 12.9.

Figure 12.9: A Computer System Acting as a Facade

Similarly, in programs also you can have facades. The C# code given in Code Snippet 4 shows an example of the facade pattern.

Code Snippet 4:

```
class CPU
{
    public void startCPU() { .... }
}

class Memory
{
    public void loadMemory() {  ... }
}

class HardDisk
{
    public byte[] readHD() {  ... }
}

// creating the Facade

class MyComputer
{
    private CPU c1;
    private Memory m1;
    private HardDisk hd1;

    public MyComputer()
   {
        this.c1 = new CPU();
        this.m1 = new Memory();
        this.hd1 = new HardDisk();
    }
```

```
    public void startComputer()
    {
        c1.startCPU();
        m1.loadMemory();
        hd1.readHD();
    }
}

// creating the user

class Person
{
    static void Main()
    {
        MyComputer facade = new MyComputer();  // line1
        facade.startComputer();
    }
}
```

In Code Snippet 4, there are five classes namely, `CPU`, `Memory`, `HardDisk`, `MyComputer`, and `Person`. The constructor of the class `MyComputer` is used to initialize the objects of classes `CPU`, `Memory`, and `HardDisk`. It also consists of a method named `startComputer()` that is used to invoke the startup methods of the three classes. Now, the class `Person` need not create separate objects of the three classes. It just creates the object of class `MyComputer` as shown in line1 and invokes the `startComputer()` method, which in turn will invoke the startup methods of the other three classes. Thus, the `MyComputer` class became a façade through which the `Person` class could solve its purpose without having to concern about the internal implementation of the classes `CPU`, `Memory`, and `HardDisk`.

The facade pattern can be used in the following cases:

- To reduce the complexity of a system.
- To decouple the subsystems, reduce dependency, and improve code portability.
- To create an entry point to the subsystems.
- To improve performance of the system.
- To secure subsystem components from unauthorized clients.

Summary

- A pattern is a previously used solution to a problem that can be applied to a new program by making relevant changes.
- The factory pattern refers to a factory of objects of classes in which a factory class or factory method is used to create different type of objects.
- The singleton pattern is one that allows creation of only one instance of a class.
- The composite pattern arranges components into a tree structure consisting of individual objects as well as composite objects.
- The decorator pattern is used to dynamically add or remove responsibilities to an object without affecting other objects.
- The proxy pattern uses a simple object to depict a complex one.
- The facade pattern creates an additional layer or class like a mask to make a complex system simpler.

Check Your Progress

1. _____ pattern allows creation of different types of objects based on a user's requirement.

A)	Decorator pattern
B)	Factory pattern
C)	Facade pattern
D)	Singleton pattern

2. Match the following scenarios with their corresponding design patterns that can represent them.

	Scenario		Design Pattern
A)	Factory manufacturing goods	**1)**	Proxy pattern
B)	Librarian in a library	**2)**	Decorator pattern
C)	ATM of a bank	**3)**	Facade pattern
D)	Ice cream with nuts and honey	**4)**	Factory pattern

3. Mary is a developer in Sein Microsystems. She has been assigned a task to create a student registration form for a school that consists of different controls to input and submit data to the system. Which design pattern should she use to achieve this?

A)	Factory pattern
B)	Decorator pattern
C)	Facade pattern
D)	Composite pattern

4. Which of the following statements about the facade pattern are true?

A)	Facade pattern helps to reduce complexity of the system.
B)	Facade pattern helps to create an entry point to the subsystems.
C)	Facade pattern reduces performance of the system.
D)	Facade pattern makes subsystem components vulnerable to unauthorized access.

Check Your Progress

5. Match the following patterns with their corresponding descriptions.

	Pattern		Description
A)	Singleton	**1)**	Creates an external mask to make a complex system simpler.
B)	Decorator	**2)**	Arranges components into a tree structure.
C)	Composite	**3)**	Allows creation of only one instance of a class.
D)	Facade	**4)**	Dynamically adds or removes responsibilities to an object.

6. Which of the following options is not a use of proxy pattern?

A)	When creating an object is too expensive in terms of time or memory.
B)	To provide an alternative to sub-classing.
C)	When access permission is required to operate a complex system.
D)	To postpone creation of an object until it is actually needed.