**900220-000-00-KM-03, Principles of Object Orientated Programming with C++, NQF Level 4, Credits 6**

**Learner Guide**

**Module One (1)**

|  |  |
| --- | --- |
| **Module Code** | 900220-000-00-KM-01 |
| **NQF Level** | 4 |
| **Credits** | 6 |
| **Skills Programme ID Number** | SP- 230374 |
| **Curriculum Title** | C++ Programmer |
| **Curriculum Code** | 900220-000-00-00 |

|  |  |
| --- | --- |
| **Name** |  |
| **Contact Address** |  |
| **Telephone (H)** |  |
| **Telephone (W)** |  |
| **Facsimile** |  |
| **Cellular** |  |
| **E-mail** |  |

**Note to the learner**

This Learner Guide provides a comprehensive overview of the module. It is designed to improve the skills and knowledge of learners, and thus enabling them to effectively and efficiently complete specific tasks.

**Purpose of the Module**

The main focus of the learning in this knowledge module is to build an understanding of the principles of Object Orientated Programming with C++ and when/how to use them

The learning will enable learners to demonstrate an understanding of:

* KM-03-KT01: Object-Oriented Programming (OOP) in C++30%
* KM-03-KT02: Object-Oriented Programming in C++ concepts 30%
* KM-03-KT03: OOP Capabilities 40%

**Provider Accreditation Requirements for the Knowledge Module**

**Physical Requirements:**

* The provider must have lesson plans and structured learning material or provide learners with access to structured learning material that addresses all the topics in all the knowledge modules as well as the applied knowledge in the application.
* QCTO/ MICT SETA requirements

**Human Resource Requirements:**

Lecturer/learner ratio of 1:20 (Maximum)

* Qualification of lecturer (SME): o NQF 5 qualified in industry recognised qualifications with 1-year experience in the IT industry o Cybersecurity vendor certification
* Assessors and moderators: accredited by the MICT SETA

**Legal Requirements:**

* Legal (product) licences to use the software for learning and training
* OHS compliance certificate

**Venue, Date and Time:**

Consult your facilitator should there be any changes to the venue, date and/or time.Refer to your timetable.

**Assessments**

**Integrated Formative Assessment:** The skills development provider will use the curriculum to guide them on the stipulated internal assessment criteria and weighting. They will also apply the scope of practical skills and applied knowledge as stipulated by the internal assessment criteria. This formative assessment leads to entrance into the integrated external summative assessment.

**Integrated Summative Assessment**: An external integrated summative assessment conducted through the relevant QCTO Assessment Quality Partner is required to issue this qualification. The external integrated summative assessment will focus on the exit level outcomes and associated assessment criteria.

**Skills Programme Purpose**

The need for this skills programme was identified after realising the importance and future impact of the 4IR on the economy of South Africa and its competitiveness. The Minister of Communications then gazetted the Presidential Commission on the Fourth Industrial Revolution (PC4IR) on 9 April 2019. In March 2020 this Commission delivered a report with wide ranging recommendations for Human Capital Development that will drive the 4IR forward.

This report clearly indicated the speed at which companies will have to invest in big data analysis, web-enabled market investment and the use of cloud computing and machine learning. Programming skills and being competent in the use of programming languages such as C++ Language are central to these initiatives.

The development of this C++ Programmer Skills Programme is also in support of the drivers for economic recovery as stated in the Economic Reconstruction and Recovery Plan (ERRP) and the subsequent Economic Reconstruction and Recovery Skills Strategy.

**Skills Programme Purpose**

A C++ Programmer will be able to Implement solutions to solve real life problems in an efficient manner applying a knowledge and understanding of the principles of programming with C++ and applicable tools.

Tasks that the learner will be able to know, do and understand after achievement of the skills programme include:

* Create well-written and readable C++ programs, using a disciplined coding style, including documentation and indentation standards.
* Work collaboratively in a team and execute version control

**Entry Requirements**

Grade 11 with Maths Lit and English.

Access to equipment, internet connectivity and how to work remotely

**EXIT LEVEL OUTCOMES**

**Exit Level Outcomes (ELO) 1**

Describe the basics of C++ Programming

Associated Assessment Criteria

* Fundamentals of the C++ programming language are explained.
* Basic concepts and methods of C++ object-oriented programming and object-oriented design are described.
* The development life cycle as a means of creating C++ applications is described.
* A thorough knowledge of the use of algorithms in problem solving is demonstrated.

**Exit Level Outcomes (ELO) 2**

Programme effectively using C++ frameworks and functionalities

Associated Assessment Criteria

* The use of C++syntax is demonstrated by creating neat and concise coding including application of documentation and indentation standards.
* Well-written and readable C++ programs are created, using a disciplined coding style, including comments and indentation standards.
* Procedural and object oriented concepts and syntax are applied.
* The ability to troubleshoot problems with application development is demonstrated and application is debugged.

**Exit Level Outcomes (ELO) 3**

Work collaboratively in a team using the GitHub platform

Associated Assessment Criteria

* An ability to work with Git and GitHub functionalities is demonstrated.
* The ability to work collaboratively in a team using Git is applied
* Version control is executed using Git functionalities such as repositories, branches, commits and pull requests

**Session 1:** **KM-03-KT01: Object-Oriented Programming (OOP) in C++30%**

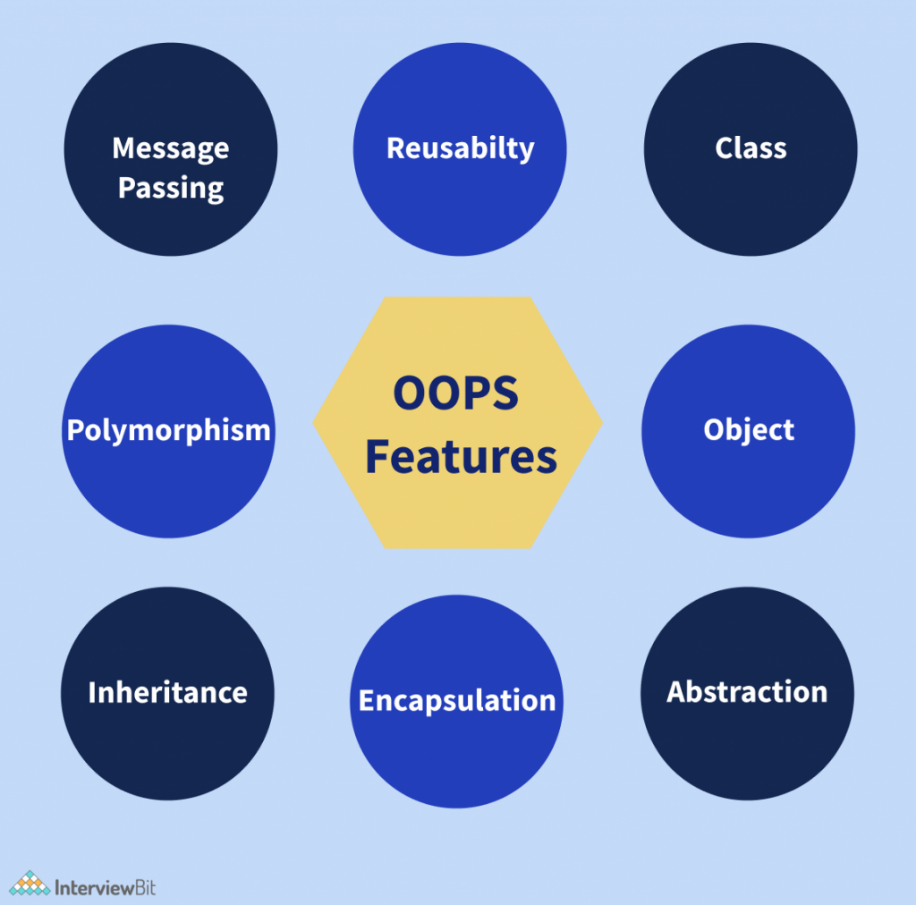
Topic elements to be covered include:

* KT0101 Concept, definition and characteristics of OOP
* KT0102 OOP paradigm
* KT0103 Four OOP principles
* KT0104 Benefits of OOP
* KT0105 OOP terminology (class, attribute, property, field, data member, class variables, etc.)
* KT0106 Building blocks of OOP:
  + - Classes
    - Objects – properties and types
    - Methods
    - Attributes
* KT0107 Objects as a type of variable
* KT0108 Properties and property values
* KT0109 Advantages of OOP

**KT0101 Concept, definition and characteristics of OOP**

Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of "objects." It is one of the most widely used programming paradigms and is known for its ability to model real-world entities and their interactions in a software system. Here are the key concepts, definitions, and characteristics of OOP:

* **Objects:** Objects are the fundamental building blocks of OOP. They represent real-world entities or concepts in the software. An object is a self-contained unit that combines both data (attributes or properties) and behavior (methods or functions) that operate on the data.
* **Classes:** A class is a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will have. In other words, a class provides the blueprint for creating multiple instances (objects) with the same attributes and methods.
* **Encapsulation:** Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit called an object. It hides the internal state of an object from the outside and provides controlled access through methods, which are known as accessors and mutators.
* **Inheritance:** Inheritance is a mechanism that allows one class (the subclass or derived class) to inherit the properties and behaviors of another class (the superclass or base class). This promotes code reuse and the creation of hierarchies of related classes.
* **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the use of a single interface to represent different types of objects. Polymorphism is often achieved through method overriding and interfaces in languages like Java and C#.

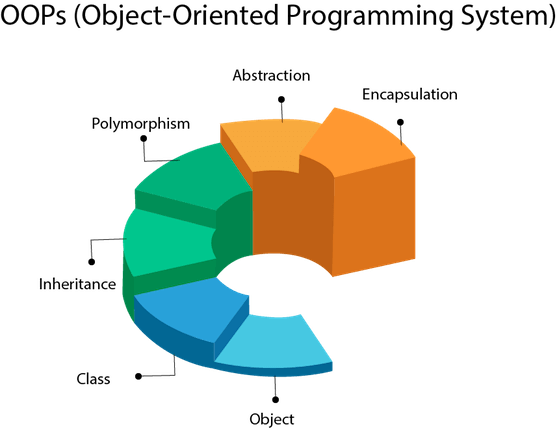


* **Abstraction:** Abstraction is the process of simplifying complex systems by breaking them down into smaller, more manageable parts. In OOP, classes and objects provide a way to abstract real-world entities and their interactions, allowing developers to focus on essential details while hiding unnecessary complexity.
* **Message Passing:** Objects in OOP communicate with each other by sending and receiving messages. This communication is often achieved through method calls, allowing objects to interact and collaborate in a modular and organized manner.
* **Modularity:** OOP promotes modularity by encapsulating data and behavior within objects. This makes it easier to develop and maintain large-scale software systems as changes to one part of the system are less likely to affect other parts.
* **Reusability:** OOP encourages code reuse through inheritance and the creation of reusable classes and components. This reduces development time and effort by leveraging existing code.
* **Flexibility and Scalability:** OOP provides a flexible and scalable way to design and build software systems. New classes and objects can be added to extend functionality without affecting existing code.

**KT0102 OOP paradigm**

Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of "objects." It is one of the most widely used programming paradigms in software development. In OOP, software is organized into objects, which are instances of classes. Here are some key principles and concepts of the OOP paradigm:

* **Objects**: Objects are instances of classes and represent real-world entities or concepts. They encapsulate both data (attributes or properties) and behavior (methods or functions). For example, you can have a "Car" class, and instances of this class would represent specific cars.
* **Classes**: Classes are blueprints or templates for creating objects. They define the structure and behavior that objects of that class will have. A class can include attributes (data members) and methods (functions).
* **Encapsulation**: Encapsulation is the concept of bundling data and methods that operate on that data into a single unit called an object. It hides the internal state of an object from the outside world and exposes a controlled interface to interact with the object. This helps in data protection and reduces the risk of unintended interference with an object's internal state.
* **Inheritance**: Inheritance allows a new class (subclass or derived class) to inherit the properties and behaviors of an existing class (superclass or base class). It promotes code reuse and the creation of a hierarchy of related classes. Subclasses can override or extend the behaviour of their parent classes.



* **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables method calls and operations to be determined at runtime based on the actual type of the object. Polymorphism is often achieved through method overriding and interfaces or abstract classes.
* **Abstraction**: Abstraction is the process of simplifying complex reality by modeling classes based on the essential characteristics of an object while ignoring non-essential details. It helps in managing complexity and creating a clear and understandable representation of objects and their interactions.
* **Message Passing**: In OOP, objects communicate with each other by sending messages. Messages are typically method calls, and the receiving object processes the message by invoking the appropriate method. This concept is fundamental to the way objects interact in an OOP system.
* **Encapsulation of State**: In OOP, an object's state (attributes or properties) is encapsulated within the object itself, and access to that state is controlled through methods. This helps maintain data integrity and allows for controlled modification of an object's state.

**KT0103 Four OOP principles**

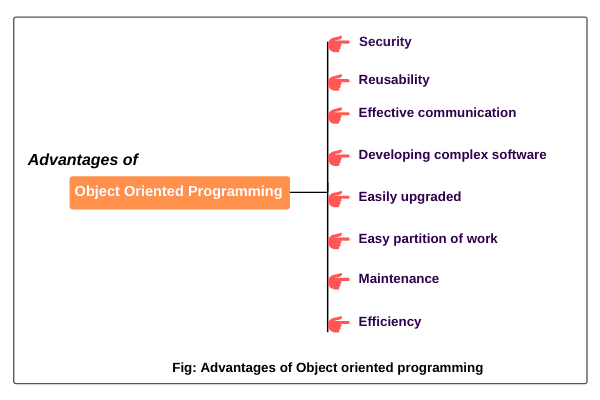
Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to organize and structure code. There are four main principles in OOP, often referred to as the "Four Pillars of OOP," which help developers create efficient, organized, and maintainable code. These principles are:

1. **Encapsulation:**
   * Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class.
   * It restricts direct access to some of the object's components, making it possible to control the state of an object through defined interfaces.
   * Access modifiers like public, private, and protected are used to control the visibility and accessibility of class members.
2. **Inheritance:**
   * Inheritance is a mechanism that allows a new class (derived or child class) to inherit properties and behaviour’s (attributes and methods) from an existing class (base or parent class).
   * It promotes code reusability and allows you to create a hierarchy of classes, where more specific classes inherit from more general ones.
   * Inheritance establishes an "is-a" relationship between classes. For example, a "Cat" class can inherit from an "Animal" class because a cat is a type of animal.
3. **Polymorphism:**
   * Polymorphism means "many forms" and refers to the ability of objects of different classes to be treated as objects of a common superclass.
   * It allows you to write code that can work with objects of multiple classes in a consistent way.
   * Polymorphism is often achieved through method overriding (in the context of inheritance) and through interfaces or abstract classes.
4. **Abstraction:**
   * Abstraction is the process of simplifying complex reality by modeling classes based on the essential properties and behaviors of objects.
   * It involves defining a class's interface with a focus on what the class does rather than how it does it. Details of implementation are hidden.
   * Abstraction allows you to manage the complexity of software systems by breaking them down into manageable, self-contained units.

**KT0104 Benefits of OOP**

Object-Oriented Programming (OOP) is a popular programming paradigm that provides several benefits for software development. Here are some of the key advantages of OOP:

* **Modularity**: OOP promotes modularity by organizing code into reusable and self-contained objects. This makes it easier to develop, test, and maintain software, as changes to one part of the codebase are less likely to affect other parts.
* **Reusability**: OOP encourages the creation of classes and objects, which can be reused in different parts of an application or in other applications altogether. This reduces duplication of code and promotes a more efficient development process.
* **Encapsulation**: OOP allows you to encapsulate data and behavior within objects. This means that the internal details of an object are hidden from the outside world, and access to the object's data is controlled through well-defined interfaces. This helps in data protection and reduces the chances of unintended interference with the object's internal state.
* **Abstraction**: Abstraction is a fundamental concept in OOP, where you can create abstract classes and interfaces that define a set of methods without providing an implementation. This allows for high-level modeling and helps in designing complex systems by focusing on essential features and ignoring irrelevant details.
* **Inheritance**: Inheritance is a mechanism in OOP where a new class (subclass or derived class) can inherit properties and behaviors from an existing class (superclass or base class). This promotes code reuse and the creation of a hierarchical structure of classes, making it easier to represent real-world relationships in software.
* **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. This enables you to write more flexible and extensible code, as you can work with objects in a generic way without knowing their specific types. Polymorphism is often achieved through method overriding and interfaces.

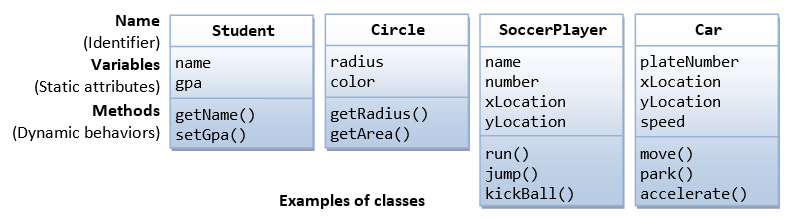


* **Maintenance and Scalability**: OOP's modular and organized structure makes it easier to maintain and extend software systems over time. When requirements change or new features are added, you can often make changes within the relevant classes and objects without affecting the entire codebase.
* **Readability and Understandability**: OOP promotes a natural way of modeling real-world entities, making the code more readable and understandable. Objects and their interactions often map closely to real-world concepts, making it easier for developers to comprehend and work with the code.
* **Team Collaboration**: OOP allows for a division of labor among development teams. Different teams can work on different classes or modules independently, as long as they adhere to the agreed-upon interfaces. This facilitates parallel development and collaboration among team members.
* **Code Organization**: OOP encourages a structured and organized approach to code organization. Classes and objects help in breaking down complex systems into manageable components, which makes it easier to design, document, and maintain large software projects.

**KT0105 OOP terminology (class, attribute, property, field, data member, class variables, etc.)**

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to structure and organize code. In OOP, there are several important terminology and concepts:

* **Class**: A class is a blueprint or template for creating objects. It defines the structure and behavior of objects that belong to it. Classes are like a blueprint for creating objects.
* **Object**: An object is an instance of a class. It is a concrete realization of the class blueprint, with its own unique data and state. Objects have both attributes and methods.
* **Attribute**: An attribute (also known as a field or data member) is a variable that belongs to an object or class. It represents some characteristic or property of the object. Attributes define the state of an object.
* **Property**: A property is a special type of attribute that often has associated getter and setter methods. Properties are used to control access to the object's data and can provide validation or other logic when getting or setting values.



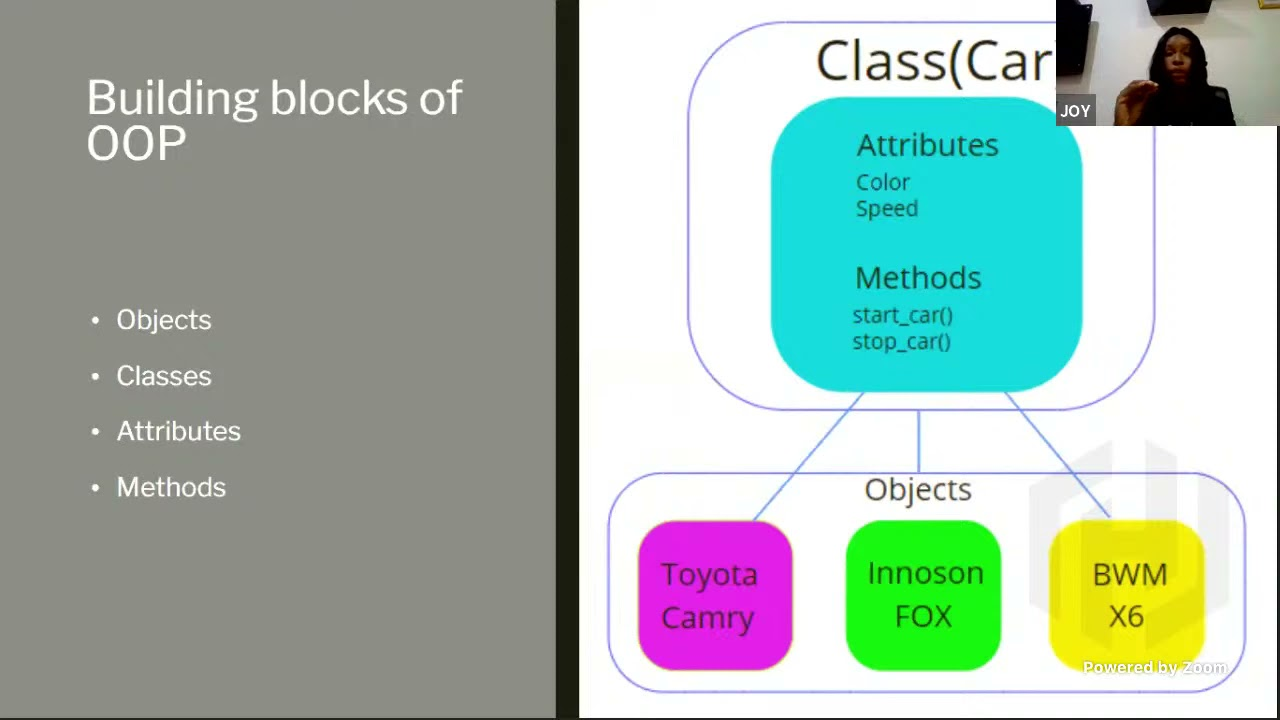
* **Field**: A field is another term for an attribute or data member. It's a variable that holds data within a class or object.
* **Data Member**: A data member is a synonym for an attribute or field. It refers to a variable that holds data within an object or class.
* **Class Variables (Static Variables)**: Class variables are variables that are shared among all instances (objects) of a class. They are defined at the class level and not at the instance level. Changes to a class variable affect all instances of the class.
* **Instance Variables (Instance Fields)**: Instance variables are variables that are unique to each instance of a class. They hold data specific to that instance and can have different values for different objects of the same class.
* **Method**: A method is a function that is associated with a class and can be called on objects of that class. Methods define the behavior of objects.
* **Constructor**: A constructor is a special method used for initializing objects when they are created. It typically sets the initial state of the object by assigning values to its attributes.
* **Encapsulation**: Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit called a class. It provides data hiding and abstraction, allowing the internal implementation details to be hidden from the outside.
* **Inheritance**: Inheritance is a mechanism in OOP that allows a class (subclass or derived class) to inherit properties and behaviors (attributes and methods) from another class (superclass or base class). It promotes code reuse and the creation of hierarchical relationships between classes.
* **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the use of a single interface to represent a general class of actions. Polymorphism can be achieved through method overriding and interfaces/abstract classes.

**KT0106 Building blocks of OOP:**

**Classes**

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes as the fundamental building blocks for designing and organizing code. Classes are templates or blueprints for creating objects, and they encapsulate the behavior and data of those objects. Here are the key building blocks of OOP classes:

* **Class**: A class is a blueprint or template for creating objects. It defines the structure and behavior that its objects will have. In a class, you specify attributes (also known as properties) and methods (functions or behaviors).
* **Object**: An object is an instance of a class. It represents a concrete, individual entity that has its own state and behavior based on the class's definition. Objects are created from a class using a process called instantiation.
* **Attributes (Properties)**: Attributes are data members or fields that define the characteristics or properties of an object. These can be variables that store data specific to each object created from the class.



Attributes describe the object's state.

class Car:

def \_\_init\_\_(self, make, model):

self.make = make # 'make' is an attribute

self.model = model # 'model' is an attribute

Methods (Behaviors): Methods are functions defined within a class that specify the actions or behaviors that objects of that class can perform. Methods operate on the object's attributes and can change the object's state.

python

class Car:

def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

def start\_engine(self):

print(f"The {self.make} {self.model}'s engine is now running.")

Constructor: The constructor is a special method (usually named \_\_init\_\_ in Python) that is automatically called when an object is created from a class. It initializes the object's attributes with initial values.

python

def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

Inheritance: Inheritance is a mechanism that allows one class (the subclass or derived class) to inherit the attributes and methods of another class (the base class or parent class). This promotes code reuse and the creation of hierarchies of related classes.

python

class ElectricCar(Car): # ElectricCar is a subclass of Car

def \_\_init\_\_(self, make, model, battery\_capacity):

super().\_\_init\_\_(make, model)

self.battery\_capacity = battery\_capacity

Encapsulation: Encapsulation is the concept of bundling data (attributes) and methods (behaviors) that operate on that data into a single unit (a class). It hides the internal details of how the class works, allowing for data abstraction and control over access to the data through methods.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common base class. This enables the use of a consistent interface for different types of objects, making code more flexible and extensible.

Abstraction: Abstraction is the process of simplifying complex reality by modeling classes based on relevant attributes and behaviors while ignoring unnecessary details. It helps in managing the complexity of software systems.

**Objects – properties and types**

In Object-Oriented Programming (OOP), objects are the fundamental building blocks. Objects are instances of classes and represent real-world entities in your program. Objects have properties and types, which are essential aspects of how they are defined and used. Let's explore these building blocks of OOP objects:

1. **Properties (Attributes)**:

Properties, also known as attributes or fields, represent the characteristics or data associated with an object. These properties define the object's state. Each object of a class has its own set of property values. Properties can have different data types, such as integers, strings, booleans, or even other objects.

For example, in a class representing a "Person," properties might include "name," "age," and "address":

class Person: def \_\_init\_\_(self, name, age, address): self.name = name # 'name' is a property self.age = age # 'age' is a property self.address = address # 'address' is a property

In this example, the properties of a "Person" object store the name, age, and address of that person.

1. **Types**:

The type of an object refers to the class it belongs to. In statically-typed languages like Java or C++, the type of an object is explicitly declared. In dynamically-typed languages like Python, the type is determined at runtime.

For instance, if you create an object from a class "Car," that object's type is "Car." If you create another object from a class "Bicycle," its type is "Bicycle."

car\_obj = Car() # The type of car\_obj is "Car" bicycle\_obj = Bicycle() # The type of bicycle\_obj is "Bicycle"

Types are essential for determining which methods and properties are available for an object, as different classes may have different behaviors and attributes.

In summary, objects in OOP have properties (attributes) that define their state and types that define their class membership. Properties represent the data associated with an object, while types determine the class from which the object was instantiated. These properties and types are key elements in creating and working with objects in an object-oriented programming paradigm.

**Methods**

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes. Methods are an essential part of OOP, as they define the behavior or actions that objects can perform. Here are the building blocks of methods in OOP:

1. Class: A class is a blueprint or a template for creating objects. It defines the attributes (data members) and methods (functions) that the objects of the class will have. Methods are typically declared within a class.
2. Method Declaration: In a class, methods are declared with a specific syntax. A method declaration includes the method's name, return type, and parameters (if any). For example:

class MyClass: def my method(self, param1, param2): # method implementation

In this example, my\_method is a method of the MyClass class.

1. Method Signature: The method signature consists of the method's name and its parameter list. It uniquely identifies a method within a class. Overloading is a feature in some OOP languages that allows multiple methods with the same name but different parameter lists (method overloading).
2. Access Modifiers: Access modifiers determine the visibility or accessibility of methods. Common access modifiers include public, private, and protected. These modifiers control who can call or access the method. For example, in Python, methods are typically public by default unless explicitly marked as private.
3. Return Type: Methods often have a return type, indicating the type of value they return after performing their operations. Methods can have void return types, meaning they do not return any value. Other return types can include integers, strings, custom objects, etc.
4. Method Implementation: This is the actual code or instructions inside the method. It defines what the method does when it is called. The implementation can include statements, calculations, conditionals, and more.
5. Parameters: Methods can take parameters or arguments that provide input data to the method. Parameters are defined in the method's parameter list and can be used within the method's implementation. Parameters are used to pass information from the caller to the method.
6. Method Invocation: To execute a method, you need to create an instance (object) of the class, and then you can invoke the method using the object. For example:

my\_object = MyClass() result = my\_object.my\_method(arg1, arg2)

Here, my method is called on the my object instance with arg1 and arg2 as arguments.

1. Method Overriding: Inheritance allows classes to inherit methods from parent classes. Subclasses can override (provide their own implementation for) methods inherited from their parent class. This is a fundamental concept in polymorphism.
2. Static Methods: Some languages support static methods that are not associated with any specific instance of a class. Static methods can be called on the class itself, rather than on an instance of the class.

These building blocks collectively enable you to define the behavior of objects in an object-oriented program by encapsulating data and functions into classes and methods. OOP promotes code reusability, modularity, and a structured way of organizing code.

**Attributes**

In Object-Oriented Programming (OOP), attributes, also known as properties or fields, are the building blocks that represent the characteristics or data associated with an object. These attributes define the state of an object and provide the data that the object can work with. Here are some key concepts related to attributes in OOP:

* Instance Variables/Attributes: These are attributes that belong to a specific instance of a class. Each object created from a class has its own set of instance variables, and their values can vary from one object to another.
* Class Variables/Attributes: Class attributes are shared by all instances of a class. They are defined at the class level and have the same value for every object of that class. Class attributes are often used for data that is common to all instances of a class.
* Private Attributes: In many OOP languages, attributes can have access modifiers like private, protected, or public. Private attributes are only accessible within the class they are defined in. This encapsulation helps maintain data integrity and control access to the attributes.
* Public Attributes: Public attributes are accessible from outside the class. While they provide direct access to the attribute's value, it's considered good practice to use getters and setters (methods) to access and modify them to maintain encapsulation.
* Getter Methods: Getter methods are used to retrieve the values of private or protected attributes. They provide controlled access to these attributes, allowing validation or additional logic to be applied when retrieving the data.
* Setter Methods: Setter methods are used to modify the values of private or protected attributes. Like getter methods, setter methods enable you to apply validation or other logic when setting attribute values.
* Initialization: Attributes can be initialized when an object is created using a constructor or initialization method. This ensures that objects start with a known state.

**Here's an example in Python:**

class Person: def \_\_init\_\_(self, name, age): self.name = name # Instance variable self.age = age # Instance variable def get\_name(self): return self.name def set\_age(self, new\_age): if new\_age >= 0: self.age = new\_age # Creating an object person1 = Person("Alice", 30) # Accessing attributes using getter and setter methods print(person1.get\_name()) # Output: "Alice" person1.set\_age(35) # Set the age attribute # Accessing attributes directly print(person1.age) # Output: 35

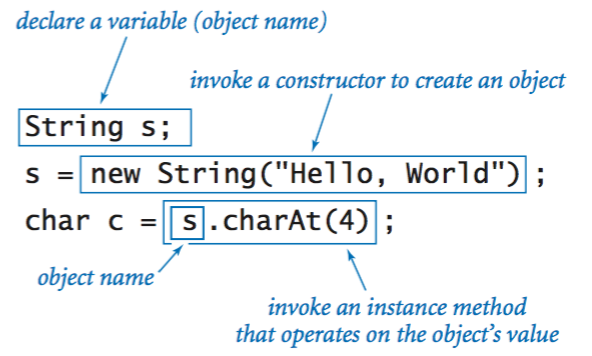
In this example, name and age are attributes of the Person class. name and age are instance variables, and get\_name() and set\_age() are getter and setter methods, respectively, for accessing and modifying these attributes.

**KT0107 Objects as a type of variable**

In computer programming, objects are not typically considered a type of variable themselves; instead, they are instances of classes or data structures. Variables, on the other hand, are used to store and manipulate data, including objects.

Here's a breakdown of the concepts:

1. Variables: Variables are named containers that can hold data. They can store various types of data, such as numbers, strings, and references to objects. Variables have a data type that specifies what kind of data they can hold. Common data types for variables include integers, floating-point numbers, strings, and more.
2. Objects: Objects are instances of classes or data structures that bundle data (attributes) and functions (methods) that operate on that data. Objects are used to model real-world entities and their behavior in object-oriented programming. Each object has a specific type, which is defined by its class.



Here's an example in Python to illustrate the relationship between variables and objects:

# Creating a class class Person: def \_\_init\_\_(self, name, age): self.name = name self.age = age # Creating objects (instances) of the Person class person1 = Person("Alice", 30) person2 = Person("Bob", 25) # Variables person1 and person2 hold references to objects

In this example, person1 and person2 are variables, and they hold references to two different objects of the Person class. The objects themselves are instances of the Person class, and they have attributes (name and age) that store data.

So, while objects are not a type of variable, variables can hold references to objects, allowing you to work with and manipulate object data through those variables.

**KT0108 Properties and property values**

Properties and property values are fundamental concepts in computer programming, web development, and data modeling. They are used to describe and store information about objects, data structures, and entities. Let's delve into these concepts in more detail:

Properties:

1. Definition: Properties refer to characteristics or attributes that describe an object, data structure, or entity. They provide information about the object's state or behavior.
2. Usage: Properties are used to store and access data associated with an object or entity. They define the structure and behavior of the object.
3. Examples: In programming, properties can represent various aspects of an object, such as a person's name, age, or address in a user object, or attributes like width and height in a graphical shape object.

Property Values:

1. Definition: Property values are the actual data or values associated with a property. They represent the specific information stored within a property for a given object or entity.
2. Usage: Property values are used to provide concrete information about the object. They can be read and modified to interact with the object's state.
3. Examples: If you have a property "name" in a user object, the property value could be "John Smith." If you have a property "width" in a rectangle object, the property value could be 10 units.

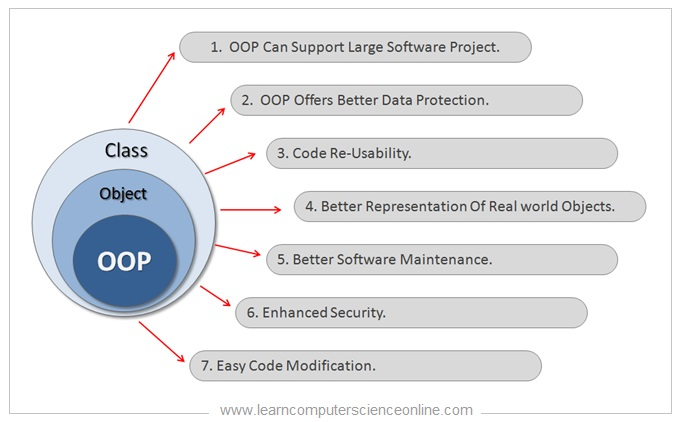
Key Characteristics:

1. Data Type: Property values often have data types associated with them, such as strings, numbers, booleans, or custom data types defined by the programmer.
2. Access and Modification: Properties allow you to access and modify their associated property values. You can get (read) the value to retrieve information or set (write) the value to update it.
3. Visibility: Some programming languages and systems may have visibility modifiers (e.g., public, private, protected) that control the accessibility of properties and their values.
4. Default Values: Properties may have default values that are assigned when an object is created unless explicitly set to a different value.
5. Validation and Constraints: In some cases, properties may have validation rules or constraints to ensure that property values meet specific criteria.
6. Inheritance: Object-oriented programming languages often support property inheritance, allowing subclasses to inherit properties from their parent classes.

**KT0109 Advantages of OOP**

Object-Oriented Programming (OOP) is a popular programming paradigm that provides several advantages for software development. Here are some of the key advantages of OOP:

* **Modularity**: OOP promotes modularity in software design. You can break down a complex system into smaller, more manageable objects. Each object encapsulates its own data and behavior, making it easier to understand, develop, and maintain.
* **Reusability**: Objects and classes can be reused in different parts of the program or even in different projects. This reusability saves time and effort in coding since you can leverage existing classes and objects to build new functionality.
* **Encapsulation**: Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit called an object. It hides the internal state and exposes a controlled interface to interact with the object. This helps in data protection and reduces the risk of unintended interference.
* **Abstraction**: Abstraction allows you to hide complex implementation details and show only the necessary features of an object. It simplifies the complexities of a system and makes it easier to work with. Developers can focus on the high-level design without getting bogged down in low-level details.
* **Inheritance**: Inheritance allows you to create new classes based on existing ones, inheriting their attributes and methods. This promotes code reuse and establishes a hierarchy of classes, which can model real-world relationships effectively.
* **Polymorphism**: Polymorphism enables you to define methods in a way that they can work with objects of different classes through a common interface. This flexibility simplifies code and makes it more adaptable to changing requirements.
* **Maintenance and Scalability**: OOP makes it easier to maintain and extend a codebase. You can make changes to individual objects or classes without affecting the entire program. This leads to better code scalability and reduces the risk of introducing bugs when making updates.



* **Collaboration**: OOP facilitates collaboration among developers. Different team members can work on different classes or objects simultaneously without interfering with each other's work, as long as they adhere to the defined interfaces and contracts.
* **Modeling Real-World Scenarios**: OOP allows you to model real-world objects, relationships, and behaviors in a way that closely resembles the problem domain. This makes it easier to understand and communicate about the software with stakeholders.
* **Code Organization**: OOP promotes a structured and organized codebase. Classes and objects provide a natural way to group related data and functionality, making the code more readable and maintainable.
* **Testing**: Object-oriented code is often easier to test because you can isolate and test individual objects or classes independently, which simplifies debugging and quality assurance processes.

**Internal Assessment Criteria and Weight**

1. IAC0101 Definitions, functions and features of Object-Oriented Programming (OOP) in C++ are understood and explained

**Session 2:** **KM-03-KT02: Object-Oriented Programming in C++ concepts 30%**

Topic elements to be covered include:

* KT0201 Basics of designing a class - class, object, state and behaviour
* KT0202 Object composition
* KT0203 Class hierarchies
* KT0204 C++ abstract class
* KT0205 C++ interface
* KT0206 C++ interface vs abstract class - a comparison
* KT0207 Introduction to abstraction
* KT0208 Introduction to C++ constructors
* KT0209 Constructor chaining
* KT0210 Initialisation of member variables

**KT0201 Basics of designing a class - class, object, state and behaviour**

Designing a class in C++ is a fundamental concept in object-oriented programming (OOP). In OOP, a class serves as a blueprint or template for creating objects. Here are the basic concepts of designing a class in C++:

1. Class:
   * A class is a user-defined data type in C++ that represents a blueprint for creating objects.
   * It defines the structure and behavior of objects of that class.
   * Classes encapsulate data (attributes) and methods (functions) that operate on that data.

Example:

class Car { public: // Member variables (attributes) std::string brand; int year; // Member functions (methods) void startEngine() { // Code to start the car's engine } void stopEngine() { // Code to stop the car's engine } };

1. Object:
   * An object is an instance of a class. It is a concrete, real-world entity created based on the class blueprint.
   * Objects have their own unique state (attributes) and can perform actions (methods).

Example:

// Creating objects of the 'Car' class Car myCar; Car anotherCar; // Setting object state (attributes) myCar.brand = "Toyota"; myCar.year = 2020; anotherCar.brand = "Honda"; anotherCar.year = 2022; // Calling object behavior (methods) myCar.startEngine(); anotherCar.stopEngine();

1. State:
   * State refers to the attributes or data associated with an object.
   * In the 'Car' class example above, brand and year are attributes that represent the state of a car object.
2. Behavior:
   * Behavior refers to the methods or functions that define what an object can do.
   * In the 'Car' class example, startEngine() and stopEngine() are methods that represent the behavior of a car object.

When designing a class in C++, consider the following principles:

* Encapsulation: Hide the internal details of the class from external code by using access modifiers (public, private, protected) to control the visibility of members. Typically, data members (attributes) are private, and methods (functions) are public.
* Abstraction: Provide a simplified and well-defined interface to interact with objects, hiding complex implementation details. Users of the class should not need to know how the class works internally.
* Inheritance: Use inheritance to create new classes based on existing ones. Inheritance allows you to reuse and extend the functionality of a base class.
* Polymorphism: Implement polymorphism through function overloading and virtual functions. Polymorphism allows different objects of the same class hierarchy to respond differently to the same method call.
* Constructor and Destructor: Provide constructors for initializing objects and destructors for cleaning up resources. Constructors are called when an object is created, and destructors are called when an object goes out of scope.
* Member Functions: Define methods that operate on the class's data members. These functions can manipulate the object's state and provide its behavior.
* Accessor and Mutator Methods: When needed, provide accessor (getter) and mutator (setter) methods to access and modify private data members.

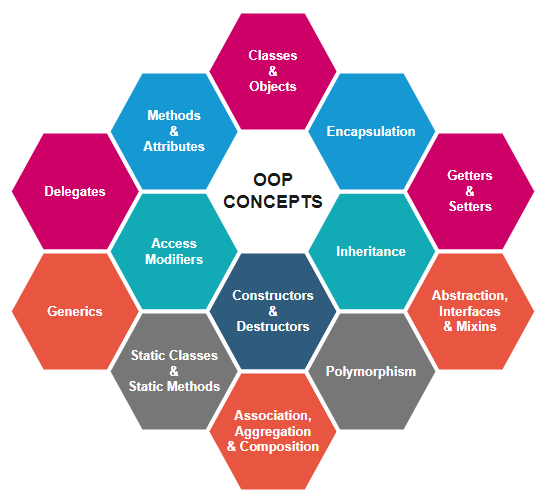
By following these principles and understanding the basic concepts of classes, objects, state, and behavior, you can effectively design and use classes in C++ for building modular and maintainable software.

**KT0202 Object composition**

Object composition is a fundamental concept in Object-Oriented Programming (OOP) that allows you to create complex objects by combining simpler objects. In C++, object composition is achieved by creating classes that have member variables that are instances of other classes. This allows you to build more complex and specialized objects by reusing existing classes and their functionality. Object composition is an alternative to inheritance, which is another way to create complex objects in OOP.

Here are the key concepts and principles of object composition in C++:

1. Classes and Objects:
   * In C++, classes are blueprints for creating objects. They define the structure and behavior of objects.
   * Objects are instances of classes. They encapsulate data (member variables) and behavior (member functions or methods).



1. Member Variables:
   * Member variables are data members of a class. They represent the state or attributes of an object.
   * You can use other classes as the data type for member variables to achieve composition.
2. Composition:
   * Composition involves creating objects of one class inside another class. This is done by declaring member variables of the inner class within the outer class.
   * The outer class is called the "containing" or "composite" class, while the inner class is called the "component" or "contained" class.
3. Encapsulation:
   * Encapsulation is one of the core principles of OOP. It means that the internal details of an object are hidden from the outside world, and access to the object's state is controlled through methods (member functions).
   * In composition, you encapsulate the component objects within the composite object, making their details hidden from external code.
4. Reusability:
   * Object composition promotes code reuse. You can create various composite objects by combining different component objects.
   * If you need to change the behavior of a component, you can do so without affecting other parts of the code that use that component.

Example of Object Composition in C++:

class Engine { public: void start() { // Code to start the engine } }; class Car { private: Engine carEngine; // Composition: Car contains an Engine object public: void startCar() { carEngine.start(); // Delegating the start operation to the Engine object // Additional car-specific logic } };

In this example, we have a Car class that contains an Engine object through composition. The Car class can use the Engine object's functionality (in this case, starting the engine) while encapsulating the details of how the engine works.

Object composition is a powerful technique for building complex systems in a modular and maintainable way, as it promotes the reuse of existing components and helps manage complexity.

**KT0203 Class hierarchies**

In Object-Oriented Programming (OOP) with C++, class hierarchies are an essential concept that allows you to create a structured and organized set of classes based on inheritance. A class hierarchy represents a relationship between classes where some classes inherit properties and behaviors from other classes. C++ uses a mechanism called inheritance to establish these relationships. Here are some key concepts related to class hierarchies in C++:

1. **Inheritance:**
   * Inheritance is a fundamental feature of OOP that allows a class (called a subclass or derived class) to inherit properties and behaviors (data members and member functions) from another class (called a superclass or base class).
   * In C++, you specify inheritance using the class or struct declaration, followed by a colon (:) and the access specifier (public, protected, or private) followed by the name of the base class.

class Base { public: int baseVar; void baseFunction(); }; class Derived : public Base { public: int derivedVar; void derivedFunction(); };

1. **Base Class:**
   * A base class (or superclass) is the class from which other classes inherit properties and behaviors. It serves as a template for creating derived classes.
   * Base classes can define common functionality that is shared among multiple derived classes.
2. **Derived Class:**
   * A derived class (or subclass) is a class that inherits properties and behaviors from a base class. It can also have additional data members and member functions of its own.
   * Derived classes can extend or modify the behavior of the base class by overriding or adding new member functions.
3. **Access Specifiers:**
   * In C++, you can use access specifiers like public, protected, and private to control the visibility of inherited members in the derived class.
   * public: Inherited members have public access in the derived class.
   * protected: Inherited members have protected access in the derived class.
   * private: Inherited members have private access in the derived class.
4. **Polymorphism:**
   * Class hierarchies enable polymorphism, which allows objects of different derived classes to be treated as objects of the base class.
   * Polymorphism allows you to write more flexible and reusable code through features like virtual functions and dynamic binding.
5. **Virtual Functions:**
   * Virtual functions are functions declared in a base class with the virtual keyword. They allow derived classes to provide their own implementation (override) of the function.
   * Virtual functions enable dynamic binding, which means that the appropriate function implementation is determined at runtime based on the actual type of the object.

class Base { public: virtual void someFunction(); }; class Derived : public Base { public: void someFunction() override; // Overrides the base class function };

1. **Abstract Classes:**
   * An abstract class is a class that cannot be instantiated. It often contains one or more pure virtual functions (functions with = 0) and is meant to serve as a base for other classes.
   * Derived classes of abstract classes must provide implementations for all pure virtual functions.

class AbstractBase { public: virtual void pureVirtualFunction() = 0; // Pure virtual function };

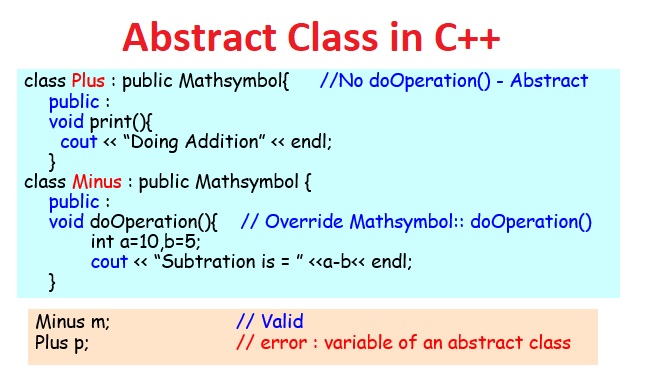
1. **Multiple Inheritance**:
   * C++ supports multiple inheritance, allowing a class to inherit from more than one base class. This can create complex class hierarchies.

class Derived : public Base1, public Base2 { // ... };

Class hierarchies are a crucial tool in C++ for modeling relationships between objects, promoting code reuse, and implementing polymorphism to create flexible and extensible software systems.

**KT0204 C++ abstract class**

In C++, an abstract class is a class that cannot be instantiated on its own and is meant to serve as a base or parent class for other classes. It is used as a blueprint for creating derived classes and typically contains one or more pure virtual functions. A pure virtual function is a function that has no implementation in the abstract class and must be overridden by any concrete (i.e., non-abstract) derived class. Abstract classes are also known as abstract base classes.



**Here's an example of how to define an abstract class in C++:**

#include <iostream> // Abstract base class class Shape { public: // Pure virtual function virtual double area() const = 0; // Pure virtual function virtual double perimeter() const = 0; }; // Concrete derived class class Circle : public Shape { private: double radius; public: Circle(double r) : radius(r) {} // Override pure virtual functions double area() const override { return 3.14159265359 \* radius \* radius; } double perimeter() const override { return 2.0 \* 3.14159265359 \* radius; } }; int main() { // Shape shape; // Error: Cannot create an instance of an abstract class Circle circle(5.0); std::cout << "Circle Area: " << circle.area() << std::endl; std::cout << "Circle Perimeter: " << circle.perimeter() << std::endl; return 0; }

In this example:

1. Shape is the abstract base class with two pure virtual functions: area() and perimeter().
2. The Circle class is a concrete derived class that inherits from Shape and provides implementations for the pure virtual functions.
3. Attempting to create an instance of the abstract class Shape directly results in a compilation error, as indicated in the commented line.
4. You can create objects of derived classes like Circle and use their overridden functions.

Abstract classes are a fundamental concept in object-oriented programming, as they allow you to define a common interface for a group of related classes while enforcing that derived classes provide specific implementations for certain methods.

**KT0205 C++ interface**

In C++, an interface is not a language feature like it is in some other programming languages such as Java or C#. However, you can achieve similar functionality using abstract classes and pure virtual functions. Here's how you can create an interface-like structure in C++:

1. **Abstract Class with Pure Virtual Functions:**

You can create an abstract base class with pure virtual functions. A pure virtual function is a function that has no implementation in the base class and must be implemented by any derived classes. This enforces derived classes to provide implementations for these functions, making it similar to an interface.

class MyInterface { public: virtual void someFunction() = 0; virtual int anotherFunction() const = 0; };

Any class that derives from MyInterface must implement both someFunction and anotherFunction.

1. **Implementing the Interface:**

Here's an example of a class implementing the MyInterface:

class MyClass : public MyInterface { public: void someFunction() override { // Implementation } int anotherFunction() const override { // Implementation return 42; } };



1. **Using the Interface:**

You can now create objects of MyClass and use them as instances of the My Interface:

int main() { MyInterface\* obj = new MyClass(); obj->someFunction(); int result = obj->anotherFunction(); delete obj; return 0; }

In this example, obj is a pointer to an instance of MyClass, but it's treated as an instance of the MyInterface, allowing you to call the interface's methods.

Keep in mind that C++ doesn't have a dedicated interface keyword like some other languages, so this approach uses abstract classes with pure virtual functions to achieve a similar effect. It's a common practice in C++ to create interfaces this way.

**KT0206** **C++ interface vs abstract class - a comparison**

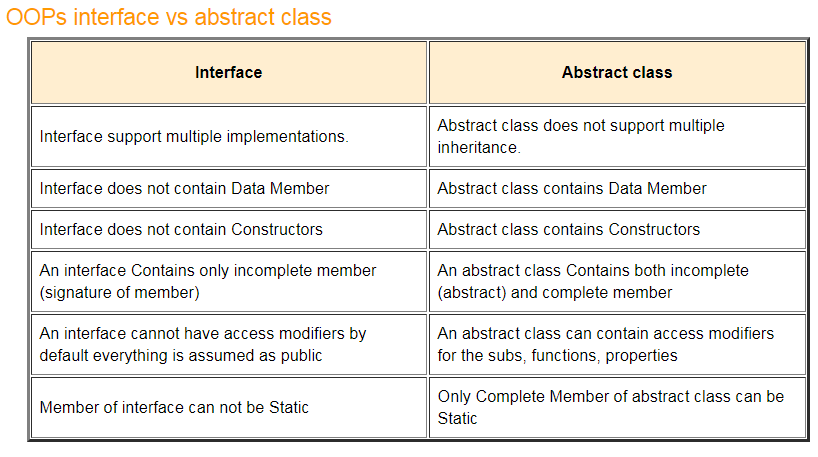
In C++, both interfaces and abstract classes are used to define contracts for classes that derive from them. They ensure that derived classes implement certain methods or provide certain functionality. However, there are some key differences between the two concepts:

1. **Syntax and Implementation:**
   * Abstract Class: An abstract class is a class that cannot be instantiated on its own. It often includes a mix of abstract (pure virtual) methods and concrete methods with implementations. Abstract methods are declared with the "virtual" keyword and the "= 0" suffix, indicating that they must be implemented by derived classes. Concrete methods can also be provided with implementations.

class AbstractBase { public: virtual void PureVirtualMethod() = 0; // Pure virtual method void ConcreteMethod() { // Implementation } };

* + Interface: An interface in C++ is a class that contains only pure virtual methods (methods without implementations) and no member variables. It is a pure contract and serves to define a set of functions that deriving classes must implement. Interfaces are created using the "class" keyword with pure virtual methods but without any member variables or concrete methods.

class InterfaceBase { public: virtual void PureVirtualMethod() = 0; // Pure virtual method };

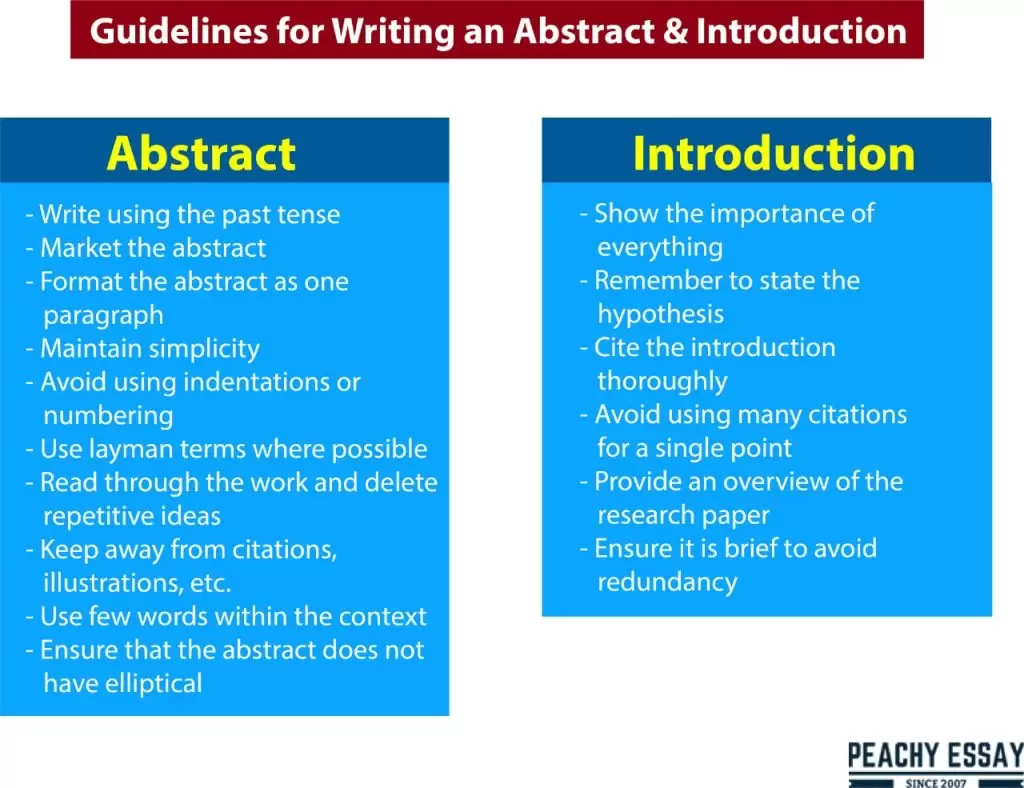


1. **Multiple Inheritance:**
   * Abstract Class: C++ allows multiple inheritance, so a class can inherit from multiple abstract classes, inheriting their functionality. However, this can lead to the "diamond problem" and can be complex to manage.
   * Interface: Interfaces are often used to achieve a form of multiple inheritance because a class can implement multiple interfaces. This allows a class to provide specific behaviors defined by different interfaces.
2. **Implementation and Design:**
   * Abstract Class: Abstract classes are useful when you want to provide some common implementation details along with the contract. Derived classes can reuse or override these concrete methods as needed.
   * Interface: Interfaces are more focused on defining a contract without any implementation details. They are used when you want to enforce a specific set of methods to be implemented by classes, without providing any default behavior.
3. **Instantiation:**
   * Abstract Class: Abstract classes cannot be instantiated on their own. They serve as a blueprint for derived classes.
   * Interface: Interfaces cannot be instantiated either. They are meant solely to define a contract that other classes must adhere to.

**KT0207 Introduction to abstraction**

Abstraction is a fundamental concept in various fields, including mathematics, computer science, philosophy, and art. It refers to the process of simplifying complex systems, ideas, or objects by focusing on essential aspects while ignoring irrelevant details. Abstraction plays a crucial role in human cognition and problem-solving and is a cornerstone of many disciplines. Here's an introduction to abstraction:

1. **Definition**: Abstraction is the process of generalizing complex real-world entities or concepts into simplified models, representations, or ideas. It involves distilling the most important aspects while omitting or simplifying less essential details.
2. **Purpose**: Abstraction serves several purposes:
   * **Simplification**: It makes complex systems or ideas more manageable and understandable.
   * **Communication**: Abstraction provides a common language for conveying ideas and concepts.
   * **Problem-solving**: It helps in solving problems by breaking them down into simpler components.
   * **Generalization**: Abstraction allows us to apply knowledge or solutions to a broader range of situations.
3. **Levels of Abstraction**:
   * **Low-Level Abstraction**: This involves focusing on fine-grained details and specific characteristics of an object or concept.
   * **High-Level Abstraction**: This concentrates on general, overarching characteristics, ignoring specific details.
4. **Examples**:
   * In mathematics, numbers and symbols are abstractions representing real-world quantities.
   * In computer science, programming languages provide abstractions for complex operations, making it easier to write code.
   * In art, abstract art represents ideas and emotions through non-representational forms and shapes.
   * In philosophy, abstract concepts like justice or morality are discussed and debated.



1. **Abstraction in Programming**:
   * Abstraction is a fundamental principle in computer programming. It allows programmers to create reusable code components, such as functions and classes, by hiding the underlying complexity.
   * Object-oriented programming (OOP) relies heavily on abstraction, where objects represent real-world entities and abstract classes define common properties and behaviors.
2. **Abstraction in Mathematics**:
   * Mathematics uses abstraction to define concepts like sets, functions, and numbers, which are foundational to various mathematical branches.
   * Mathematical abstraction helps mathematicians work with complex structures more efficiently.
3. **Challenges of Abstraction**:
   * Over-abstraction can lead to loss of important details.
   * Under-abstraction may result in a lack of clarity and understanding.
4. **Abstraction vs. Representation**: Abstraction is closely related to representation. While abstraction simplifies, representation chooses a specific way to depict an abstracted concept. For example, a map is a representation of geographical abstractions.

**KT0208 Introduction to C++ constructors**

Constructors are a fundamental concept in C++ programming and play a crucial role in object-oriented programming (OOP). They are special member functions defined within a class that are used to initialize the objects of that class. Constructors are automatically called when an object is created, and they ensure that the object is in a valid and consistent state. Here's an introduction to C++ constructors:

1. **Purpose of Constructors:**
   * Constructors initialize the attributes (data members) of an object.
   * They are responsible for allocating resources, setting default values, and performing any necessary setup for the object.
2. **Constructor Characteristics:**
   * Constructors have the same name as the class they belong to.
   * They do not have a return type, not even void.
   * Constructors are invoked automatically when an object is created.
3. **Types of Constructors:**
   * Default Constructor: A constructor with no parameters is called a default constructor. It is automatically provided by the compiler if no constructors are defined in the class. The default constructor initializes the members to their default values (e.g., zero for integers).

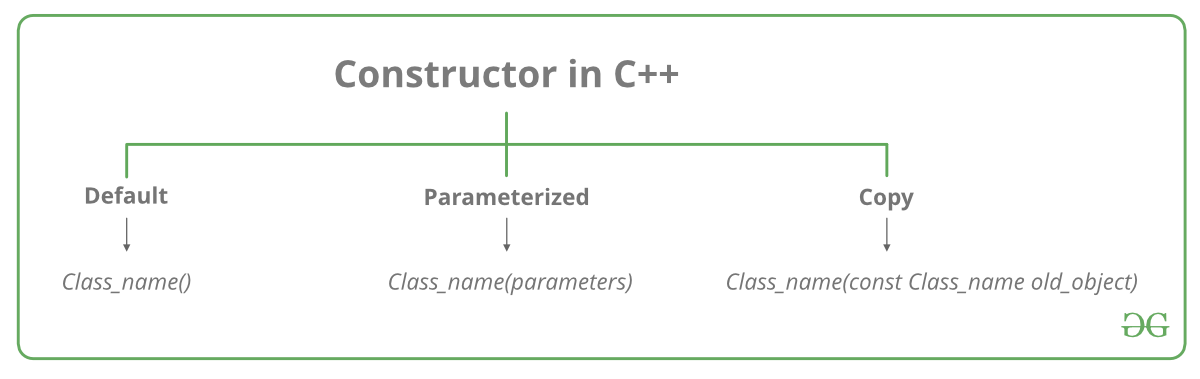
class MyClass { public: MyClass() { // Default constructor } };

* + Parameterized Constructor: Constructors can accept parameters, allowing you to initialize object attributes with specific values during object creation.

class Person { public: Person(std::string name, int age) { // Parameterized constructor this->name = name; this->age = age; } private: std::string name; int age; };

* + Copy Constructor: This constructor is used to create a new object as a copy of an existing object of the same class. It is called when one object is assigned the value of another object or when objects are passed by value to functions.

class MyClass { public: MyClass(const MyClass& other) { // Copy constructor // Initialize this object based on 'other' } };



1. **Initializer List**: Constructors can use an initializer list to initialize member variables. It's a more efficient way to initialize members, especially when dealing with complex objects.

class Rectangle { public: Rectangle(int w, int h) : width(w), height(h) { // Constructor using initializer list } private: int width; int height; };

1. **Multiple Constructors**: A class can have multiple constructors, including a mix of default, parameterized, and copy constructors, as long as they have different parameter lists. This is known as constructor overloading.

class MyClass { public: MyClass() { /\* Default constructor \*/ } MyClass(int val) { /\* Parameterized constructor \*/ } MyClass(const MyClass& other) { /\* Copy constructor \*/ } };

1. Destructors: While constructors are used for initialization, destructors are used for cleanup and resource release. They have the same name as the class, preceded by a tilde (~). A class can have only one destructor.

class MyClass { public: MyClass() { /\* Constructor \*/ } ~MyClass() { /\* Destructor \*/ } };

Constructors are a vital part of C++ classes and enable you to create and initialize objects in a well-defined manner, ensuring the integrity of your data and the proper functioning of your code.

**KT0209 Constructor chaining**

Constructor chaining is a concept in object-oriented programming where one constructor of a class can call another constructor of the same class or a superclass to initialize the object. This allows you to reuse code and ensure that the object is properly initialized regardless of which constructor is used.

In many object-oriented programming languages like Java, C++, and C#, constructor chaining is achieved using the this keyword (for the same class) or the super keyword (for a superclass). Here's a simple example in Java:

class Person { String name; int age; // Constructor with two parameters Person(String name, int age) { this.name = name; this.age = age; } // Constructor with one parameter (chaining to the two-parameter constructor) Person(String name) { this(name, 0); // Calls the two-parameter constructor with age set to 0 } // Default constructor (chaining to the one-parameter constructor) Person() { this("Unknown"); // Calls the one-parameter constructor with name set to "Unknown" } }

In this example, the Person class has three constructors. The first constructor initializes both the name and age properties. The second constructor takes only the name as a parameter and calls the first constructor with the age set to 0. The third constructor is the default constructor and calls the second constructor with "Unknown" as the name.

This way, when you create a Person object using any of the constructors, the object is properly initialized, and constructor chaining ensures that the appropriate constructor is called to set the values of the properties.

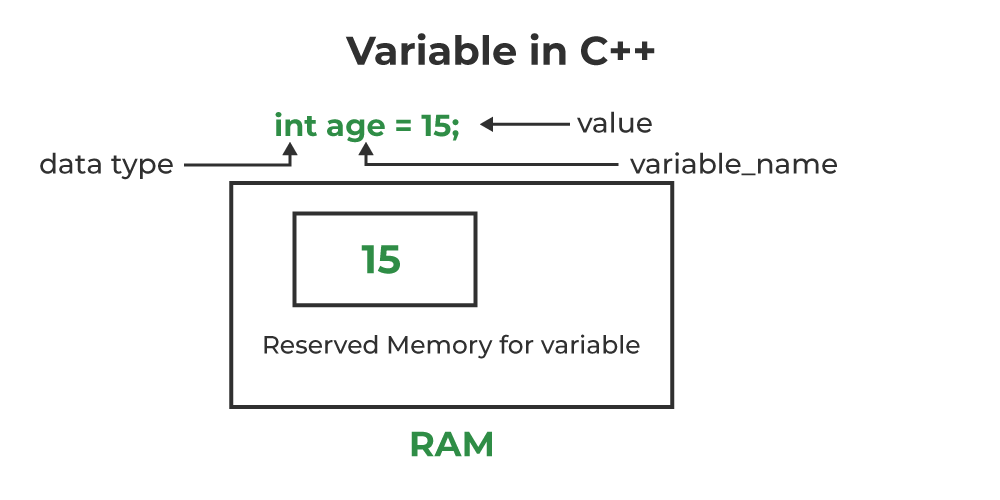
**KT0210 Initialisation of member variables**

In object-oriented programming, member variables, also known as instance variables or attributes, are variables that are associated with a class and represent the state or characteristics of objects created from that class. These member variables are typically initialized within the class constructor or using default values. Here's how you can initialize member variables in various programming languages:

1. Java: In Java, you can initialize member variables in the constructor or directly when declaring them.

public class MyClass { private int intValue; private String stringValue; // Constructor public MyClass(int intValue, String stringValue) { this.intValue = intValue; // Initialize in the constructor this.stringValue = stringValue; // Initialize in the constructor } }

1. C++: In C++, member variables can be initialized using an initializer list in the constructor.



class MyClass { private: int intValue; std::string stringValue; public: // Constructor with initialization list MyClass(int intValue, const std::string& stringValue) : intValue(intValue), stringValue(stringValue) { // Additional constructor code if needed } };

1. Python: In Python, member variables are typically initialized in the \_\_init\_\_ method of a class.

class MyClass: def \_\_init\_\_(self, intValue, stringValue): self.intValue = intValue # Initialize in the constructor self.stringValue = stringValue # Initialize in the constructor

1. C# (C Sharp): In C#, member variables can be initialized in the constructor as well.

class MyClass { private int intValue; private string stringValue; public MyClass(int intValue, string stringValue) { this.intValue = intValue; // Initialize in the constructor this.stringValue = stringValue; // Initialize in the constructor } }

1. JavaScript (ES6+): In JavaScript, you can initialize member variables in the constructor of a class.

class MyClass { constructor(intValue, stringValue) { this.intValue = intValue; // Initialize in the constructor this.stringValue = stringValue; // Initialize in the constructor } }

The key concept is that member variables should be set to appropriate initial values within the class constructor to ensure that objects created from the class have a valid initial state. Depending on the programming language, you can also set default values when declaring the variables, which will be used if no initial value is provided during object creation.

Internal Assessment Criteria and Weight

1. IAC0201 Concepts of Object-Oriented Programming (OOP) in C++ are understood

**Session 3:** **KM-03-KT03: OOP Capabilities 40%**

Topic elements to be covered include:

* KT0301 Inheritance
  + - C++ inheritance basics
    - Declaring inheritance in C++
    - Inheritance and type casting
    - Overriding methods
    - Fields and inheritance
    - Constructors and inheritance
    - Nested classes and inheritance
    - Final classes and inheritance
* KT0302 Encapsulation
  + - Encapsulation and data hiding
    - Encapsulation and abstraction
    - Basic unit of encapsulation
* KT0303 Polymorphism
  + - Polymorphism and its types
    - Compile time polymorphism
    - Runtime polymorphism
    - Pure virtual function
* KT0304 Association
  + - Aggregation and composition
    - HAS-A relationship between objects of two individual classes
* KT0305 Benefits

**KT0301 Inheritance**

**C++ inheritance basics**

In C++, inheritance is a fundamental concept that allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). Inheritance enables code reuse and allows you to model relationships between classes in an object-oriented manner. Here are some basic concepts and syntax related to C++ inheritance:

1. **Base Class (Parent Class):**
   * A base class is the class from which other classes inherit properties and behaviors.
   * It is also called the parent class or superclass.
2. **Derived Class (Child Class):**
   * A derived class is a class that inherits properties and behaviors from a base class.
   * It is also called the child class or subclass.
3. **Syntax for Declaring a Base Class:**

class BaseClass { // Base class members (data members and member functions) };

1. **Syntax for Declaring a Derived Class:**

class DerivedClass : public BaseClass { // Derived class members (additional data members and member functions) };

* + The public access specifier indicates that the derived class has public access to the base class members.

1. **Access Control:**
   * In C++, inheritance supports three access control specifiers: public, protected, and private.
     + public: Public members of the base class remain public in the derived class.
     + protected: Public and protected members of the base class become protected in the derived class.
     + private: Public and protected members of the base class become private in the derived class.
2. **Constructor and Destructor in Derived Classes**:
   * Derived classes can have their own constructors and destructors.
   * The base class constructor is called before the derived class constructor.
   * The base class destructor is called after the derived class destructor.
3. **Member Function Overriding:**
   * Derived classes can override base class member functions with their own implementations.
   * This allows for polymorphism, where the appropriate function is called based on the object's actual type.

**Here's an example illustrating these concepts:**

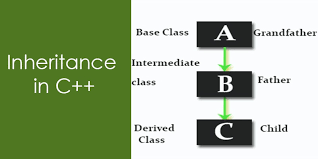
#include <iostream> class BaseClass { public: void display() { std::cout << "BaseClass display()" << std::endl; } }; class DerivedClass : public BaseClass { public: void display() { std::cout << "DerivedClass display()" << std::endl; } }; int main() { BaseClass baseObj; DerivedClass derivedObj; baseObj.display(); // Calls BaseClass's display() derivedObj.display(); // Calls DerivedClass's display() // Using a base class pointer to a derived class object BaseClass\* ptr = &derivedObj; ptr->display(); // Calls DerivedClass's display() due to polymorphism return 0; }

In this example, you can see how a derived class inherits from a base class and how member function overriding and polymorphism work when calling the display function on different objects.

**Declaring inheritance in C++**

In C++, inheritance is a fundamental feature of object-oriented programming that allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). To declare and use inheritance in C++, you can use the class or struct keyword followed by a colon (:) to specify the base class from which the derived class inherits. Here's the basic syntax for declaring inheritance:

class BaseClass { // Base class members and functions }; class DerivedClass : accessSpecifier BaseClass { // Derived class members and functions };



In this syntax:

* BaseClass is the name of the class you want to inherit from, and it's referred to as the base class or parent class.
* DerivedClass is the name of the new class you are creating, and it's referred to as the derived class or child class.
* accessSpecifier is an optional access specifier (e.g., public, protected, or private) that determines the visibility of the inherited members in the derived class. If you don't specify an access specifier, private is assumed by default.

Here's a brief explanation of the access specifiers:

* public: Inherited members remain public in the derived class, maintaining their original access levels.
* protected: Inherited members become protected in the derived class, regardless of their original access levels.
* private: Inherited members become private in the derived class, regardless of their original access levels.

**Here's an example:**

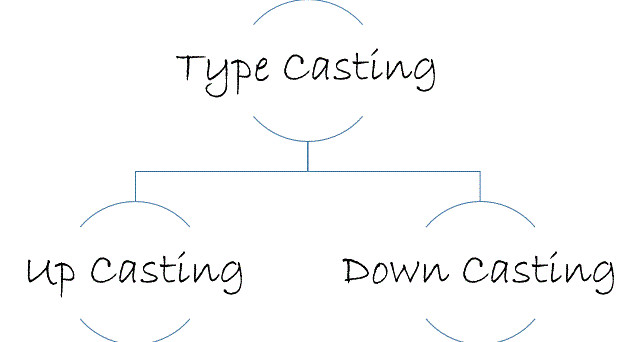
class Animal { public: void eat() { cout << "Animal is eating." << endl; } }; class Dog : public Animal { public: void bark() { cout << "Dog is barking." << endl; } }; int main() { Dog myDog; myDog.eat(); // Accessing the inherited function myDog.bark(); // Accessing the derived class function return 0; }

In this example, Dog is a derived class that inherits from the Animal base class using the public access specifier. The derived class Dog can access the eat() function from the base class Animal, and it also has its own member function bark().

**Inheritance and type casting**

Inheritance and type casting are important concepts in object-oriented programming (OOP) that play a crucial role in organizing and manipulating objects within a class hierarchy. Let's delve into each concept individually and then discuss how they relate to each other.

1. **Inheritance:**
   * Inheritance is one of the fundamental principles of OOP.
   * It allows you to create a new class (called a derived or subclass) based on an existing class (called a base or superclass).
   * The derived class inherits the properties (attributes) and behaviors (methods) of the base class.
   * Inheritance promotes code reuse and establishes a hierarchy among classes, where more specialized classes (subclasses) inherit characteristics from more general classes (superclasses).



Example:

class Animal: def speak(self): pass class Dog (Animal): def speak(self): return "Woof!" class Cat(Animal): def speak(self): return "Meow!"

In this example, Dog and Cat are subclasses of the Animal superclass. They inherit the speak method from Animal and override it to provide their own implementations.

1. **Type Casting:**
   * Type casting (or type conversion) is the process of changing the data type of an object from one class to another.
   * It is often used when you have a reference to an object of a base class, but you want to treat it as an object of a derived class.
   * Type casting allows you to access the specific properties and methods of the derived class.

In Python, type casting can be done using the isinstance () function to check an object's type and the is keyword for identity testing.

**Example:**

animal = Dog () if isinstance (animal, Dog): print (animal. speak()) # Outputs: "Woof!" # Type casting to treat 'animal' as a Cat if isinstance (animal, Cat): print(animal. speak()) # This block won't be executed # Type casting to treat 'animal' as an Animal if isinstance(animal, Animal): print(animal.speak()) # Outputs: "Woof!"

In this example, we first create an instance of the Dog class and then use type casting to treat it as an Animal and access the speak method. The isinstance () function helps ensure that the type casting is appropriate and safe.

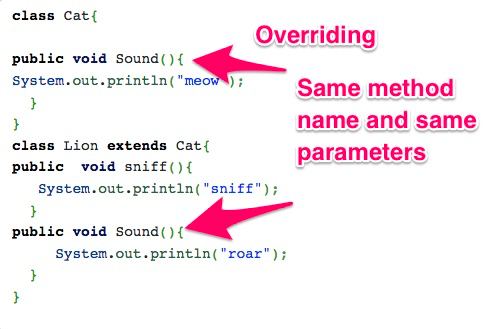
In summary, inheritance allows you to create class hierarchies and share characteristics between classes, while type casting allows you to work with objects of derived classes as if they were objects of their base classes, as long as the cast is valid based on the class hierarchy.

**Overriding methods**

In object-oriented programming (OOP), overriding methods is a concept where a subclass provides a specific implementation for a method that is already defined in its superclass. This allows the subclass to customize or extend the behavior of the inherited method without changing the method's name or signature. Overriding is a fundamental feature of polymorphism in OOP and is used to achieve runtime polymorphism.

Here are the key points to understand about method overriding:

1. Inheritance: Overriding methods is closely related to the concept of inheritance. When a subclass inherits from a superclass, it inherits all the methods and properties of the superclass, including its behavior.
2. Method Signature: The overridden method in the subclass must have the same method signature as the method in the superclass. This includes the method name, return type, and parameter types. The method name and parameters must match exactly.
3. Access Control: The access level of the overriding method in the subclass should be the same or less restrictive than the access level of the overridden method in the superclass. In many programming languages, this means you can't reduce the visibility of the method in the subclass (e.g., changing a public method to private).
4. Keyword: In most OOP languages like Java and C#, you use the @Override (Java) or override (C#) keyword to explicitly indicate that a method in a subclass is intended to override a method in the superclass. This helps catch errors at compile-time if there's a mismatch in method signatures.
5. Runtime Polymorphism: Method overriding is a way to achieve runtime polymorphism. When you call the overridden method on an object of the subclass, the method in the subclass is executed, not the one in the superclass. This allows you to write code that can work with objects of different subclasses through a common interface (the superclass).



**Here's an example in Java:**

class Animal { void makeSound() { System.out.println("Some generic sound"); } } class Dog extends Animal { @Override void makeSound() { System.out.println("Bark!"); } } public class Main { public static void main(String[] args) { Animal myDog = new Dog(); myDog.makeSound(); // Calls the overridden method in Dog class } }

In this example, we have a superclass Animal with a makeSound method, and a subclass Dog that overrides this method to provide a specific implementation. When we create a Dog object and call makeSound on it, it calls the makeSound method of the Dog class, demonstrating method overriding and runtime polymorphism.

Remember that method overriding is a powerful mechanism for creating more flexible and extensible code in OOP, allowing you to customize the behavior of objects in a hierarchical class structure.

**Fields and inheritance**

Fields and inheritance are fundamental concepts in object-oriented programming (OOP) that play a crucial role in organizing and structuring code.

1. **Fields**:
   * Fields are variables or data members within a class that hold data or state for objects of that class. They represent the characteristics or attributes of objects.
   * Fields can have different access modifiers (e.g., public, private, protected) to control their visibility and accessibility from outside the class.
   * Fields are often associated with instance variables (also known as instance fields) and class variables (also known as static fields).

public class Person { // Instance fields private String name; private int age; // Static field public static int numberOfPeople = 0; // Constructor public Person(String name, int age) { this.name = name; this.age = age; numberOfPeople++; // Increment the static field } }

1. **Inheritance:**
   * Inheritance is a mechanism in OOP that allows one class (the subclass or derived class) to inherit properties and behaviors (fields and methods) from another class (the superclass or base class).
   * It promotes code reusability and the creation of a hierarchy of classes.
   * Inheritance is represented using the "extends" keyword in many programming languages like Java, C++, and Python.

// Base class public class Animal { protected String name; public Animal(String name) { this.name = name; } public void eat() { System.out.println(name + " is eating."); } } // Derived class (inherits from Animal) public class Dog extends Animal { public Dog(String name) { super(name); // Call the superclass constructor } public void bark() { System.out.println(name + " is barking."); } }

In the above example, the Dog class inherits the name field and eat method from the Animal class. It also adds its own method, bark.

1. Access Modifiers:
   * Access modifiers (e.g., public, private, protected) control the visibility and accessibility of fields and methods in classes.
   * Public fields/methods are accessible from anywhere.
   * Private fields/methods are only accessible within the class that defines them.
   * Protected fields/methods are accessible within the class and its subclasses.

public class Example { public int publicField; private int privateField; protected int protectedField; }

In summary, fields represent the data or state within a class, while inheritance allows one class to inherit properties and behaviors from another class. Access modifiers are used to control the visibility and accessibility of fields and methods within classes and their subclasses. These concepts are fundamental in designing and organizing object-oriented code.

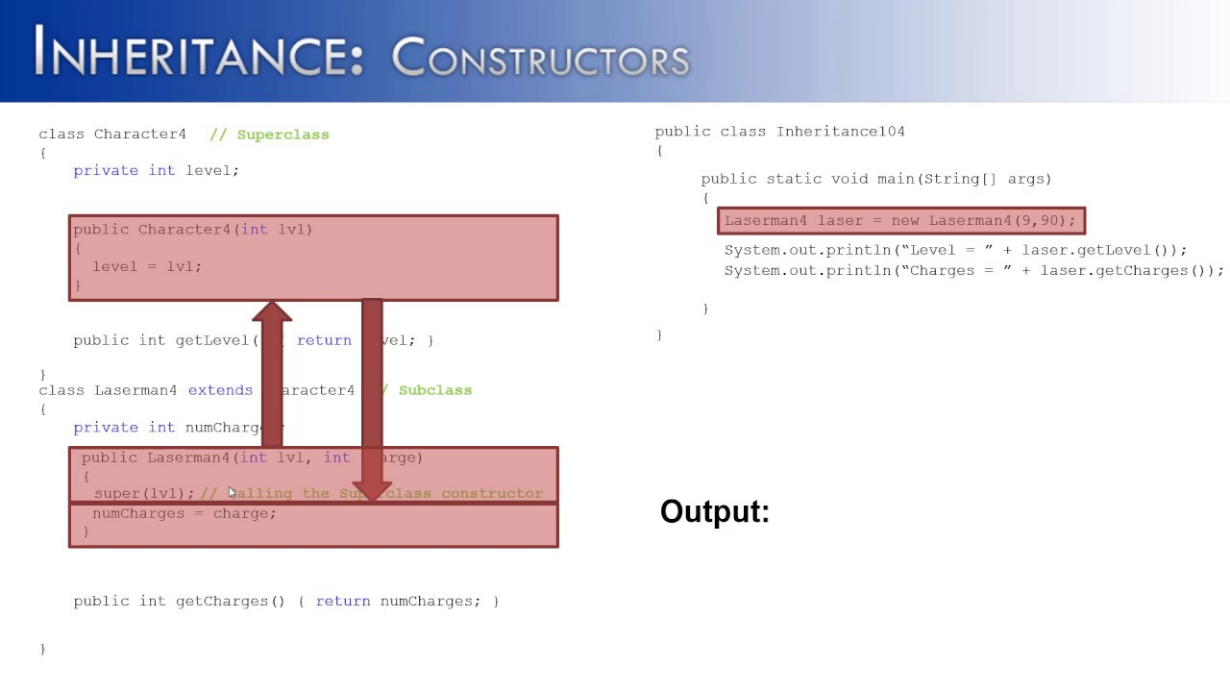
**Constructors and inheritance**

Constructors and inheritance are fundamental concepts in object-oriented programming (OOP). They play a crucial role in defining and initializing objects in a class hierarchy. Let's explore each of these concepts separately and then see how they interact:

1. **Constructors:**
   * Constructors are special methods in a class that are responsible for initializing the attributes (or properties) of an object when it is created.
   * Constructors typically have the same name as the class and are used to set initial values for instance variables.
   * In many programming languages like Java, C++, and Python, you can have multiple constructors with different parameter lists (overloaded constructors) to provide flexibility when creating objects.
   * Constructors are invoked automatically when an object of a class is created.

Here's an example of a simple constructor in Python:

class Person: def \_\_init\_\_(self, name, age): self.name = name self.age = age # Creating an instance of the Person class person1 = Person("Alice", 30)



1. **Inheritance:**
   * Inheritance is a mechanism in OOP that allows you to create a new class (called a subclass or derived class) by inheriting properties and behaviors (attributes and methods) from an existing class (called a superclass or base class).
   * The subclass can extend, override, or specialize the functionality of the superclass.
   * Inheritance promotes code reuse and the creation of a class hierarchy.

**Here's an example of inheritance in Python:**

class Animal: def \_\_init\_\_(self, name): self.name = name def speak(self): pass class Dog(Animal): def speak(self): return f"{self.name} says Woof!" class Cat(Animal): def speak(self): return f"{self.name} says Meow!" # Creating instances of the subclasses dog = Dog("Buddy") cat = Cat("Whiskers") print(dog.speak()) # Output: Buddy says Woof! print(cat.speak()) # Output: Whiskers says Meow!

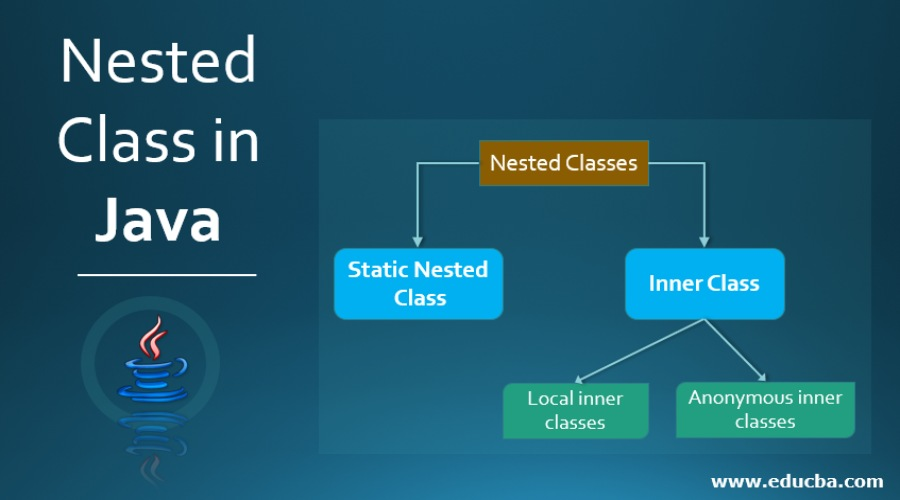
In this example, the Animal class serves as the superclass, and Dog and Cat are subclasses. They inherit the name attribute and define their own speak method.

The interaction between constructors and inheritance occurs when a subclass is created. In most programming languages, when you create an instance of a subclass, the constructor of the superclass is called automatically, followed by the constructor of the subclass. This ensures that the attributes of both the superclass and the subclass are initialized correctly.

**Nested classes and inheritance**

Nested classes and inheritance are two concepts in object-oriented programming that deal with organizing and structuring code. Let's explore each concept separately and then discuss how they can be used together.

1. **Nested Classes:**
   * **Definition:** A nested class, also known as an inner class, is a class that is defined within another class. It exists within the scope of the outer class and can access the members (fields and methods) of the outer class.
   * **Purpose:** Nested classes are often used to group related classes together and encapsulate them within a parent class. They can improve code organization and help manage complexity.
   * **Types of Nested Classes:** There are generally four types of nested classes:
     + Static Nested Class: A nested class declared as static does not have access to the instance members of the outer class and can be instantiated independently.
     + Inner Class: An inner class is not declared as static and can access both static and instance members of the outer class. It is typically used for more tightly coupled relationships.
     + Local Inner Class: A class defined within a method or block of code, with limited scope and visibility.
     + Anonymous Inner Class: A class defined without a name, often used for creating one-time objects with overridden methods.



1. **Inheritance:**
   * Definition: Inheritance is a mechanism in object-oriented programming that allows a class (the subclass or derived class) to inherit properties and behaviors (fields and methods) from another class (the superclass or base class).
   * Purpose: Inheritance promotes code reuse and allows you to model a hierarchy of classes with shared characteristics. It establishes an "is-a" relationship between the derived class and the base class.
   * Types of Inheritance: There are different types of inheritance, including single inheritance (a class inherits from one base class) and multiple inheritance (a class inherits from multiple base classes). Multiple inheritance is supported in some programming languages but not all.

Now, let's discuss how nested classes and inheritance can be used together:

Nested Classes in Inheritance: Nested classes can be used in the context of inheritance to achieve encapsulation and better organization of classes within a hierarchy. Here's how they can be used:

1. **Nested Classes as Helper Classes:** You can use nested classes to define helper classes or specialized data structures that are closely related to a particular subclass. These nested classes can be used to encapsulate implementation details and keep them within the scope of the subclass.
2. **Inner Classes for Specialization:** When a subclass extends a base class, it can define inner classes that are specific to its requirements. These inner classes can access both the superclass's and subclass's members, allowing for more specialized behavior.
3. **Static Nested Classes for Utility:** If you have utility classes that are closely related to a superclass but don't need access to instance-specific data, you can define them as static nested classes within the superclass. This can help organize the code and make it more readable.

Here's a simple Java example illustrating the use of nested classes and inheritance:

class Vehicle { // Outer class (superclass) static class Engine { // Static nested class void start() { System.out.println("Engine started."); } } class Wheel { // Inner class void rotate() { System.out.println("Wheel rotating."); } } } class Car extends Vehicle { // Subclass void drive() { System.out.println("Car is driving."); } } public class Main { public static void main(String[] args) { Car myCar = new Car(); myCar.drive(); Vehicle.Engine carEngine = new Vehicle.Engine(); carEngine.start(); Vehicle.Wheel carWheel = myCar.new Wheel(); carWheel.rotate(); } }

In this example, Engine is a static nested class, Wheel is an inner class, and Car is a subclass of Vehicle. The code demonstrates how nested classes can be used in the context of inheritance.

**Final classes and inheritance**

In object-oriented programming, final classes and inheritance are concepts that are often used to control and restrict the behavior of classes and their relationships in a programming language. Let's explore what each of these concepts means:

1. Final Classes:
   * A final class is a class that cannot be extended or inherited from. In many object-oriented programming languages like Java and C#, you can declare a class as "final" to prevent other classes from subclassing or extending it.
   * The primary reason to declare a class as final is to enforce a level of control and ensure that the class's behavior and functionality remain unchanged. This can be useful in scenarios where you want to prevent unintended modifications or subclassing that could break the class's intended functionality.
   * For example, in Java, you can declare a class as final like this:

final class MyFinalClass { // Class members and methods }

1. Inheritance:
   * Inheritance is a fundamental concept in object-oriented programming that allows one class (the subclass or derived class) to inherit properties and behaviors from another class (the superclass or base class).
   * The subclass can extend the functionality of the superclass by adding new methods and attributes or by overriding existing ones.
   * Inheritance promotes code reuse and the creation of a hierarchical relationship between classes, where more specialized classes (subclasses) inherit characteristics from more general classes (superclasses).
   * For example, in Java, you can create a subclass that extends a superclass like this:

class Superclass { // Superclass members and methods } class Subclass extends Superclass { // Subclass members and methods }

The relationship between final classes and inheritance can be summarized as follows:

* If a class is declared as final, it cannot be extended or inherited from. In other words, you cannot create a subclass of a final class.
* If a class is not declared as final, it can be extended through inheritance, allowing you to create subclasses that inherit its properties and behaviors.

**Here's an example to illustrate the concept:**

final class FinalClass { // Class members and methods } // This will result in a compilation error since you cannot extend a final class. class Subclass extends FinalClass { // Subclass members and methods }

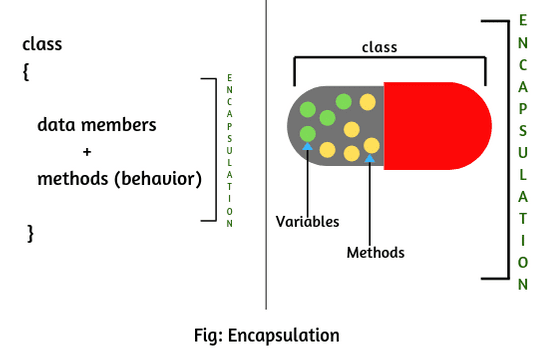
In this example, attempting to create a subclass of a final class (FinalClass) results in a compilation error.

In summary, final classes and inheritance are important concepts in object-oriented programming that allow you to control the behavior of classes and the relationships between them. Final classes restrict inheritance, while inheritance allows for the creation of subclasses that inherit properties and behaviors from a superclass.

**KT0302 Encapsulation**

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the concept of bundling data (attributes or properties) and the methods (functions or procedures) that operate on that data into a single unit called a class. This unit, also known as an object, encapsulates the data and the operations, and provides a way to control access to the internal state of an object. Here are some key aspects of encapsulation:

1. **Data Hiding:** Encapsulation hides the internal details of an object from the outside world. It restricts direct access to an object's data, allowing access only through well-defined methods or properties. This prevents unauthorized or unintended modifications to an object's state, promoting data integrity and reducing the risk of bugs.
2. **Access Control:** In most programming languages that support encapsulation, you can specify different levels of access control for the members (data and methods) of a class. Common access control modifiers include public, private, and protected. Public members are accessible from anywhere, private members are only accessible within the class itself, and protected members are accessible within the class and its subclasses.
3. **Getters and Setters:** To provide controlled access to the internal data of an object, getter and setter methods are often used. A getter allows you to retrieve the value of a private data member, while a setter allows you to modify it. By using these methods, you can implement validation logic, perform calculations, or enforce constraints when accessing or modifying data.



1. **Encapsulation Benefits:** Encapsulation enhances code maintainability, as changes to the internal implementation of a class don't affect the code that uses the class. It also promotes code reusability, as objects can be used as building blocks for more complex systems without exposing their internal workings. Additionally, encapsulation facilitates better error detection and debugging since access to data is controlled and monitored.

**Here's a simple example in Python to illustrate encapsulation**:

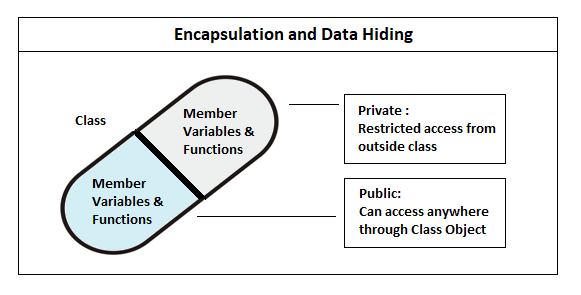
class BankAccount: def \_\_init\_\_(self, balance=0): self.\_\_balance = balance # Private attribute def deposit(self, amount): if amount > 0: self.\_\_balance += amount def withdraw(self, amount): if 0 < amount <= self.\_\_balance: self.\_\_balance -= amount def get\_balance(self): return self.\_\_balance # Create a bank account account = BankAccount(1000) # Accessing balance using a getter method print("Current balance:", account.get\_balance()) # Depositing and withdrawing money account.deposit(500) account.withdraw(200) print("Updated balance:", account.get\_balance())

In this example, the balance attribute is encapsulated within the BankAccount class, and access to it is controlled through methods (deposit, withdraw, and get\_balance). This ensures that the balance can only be modified or accessed using the defined methods, maintaining data integrity and encapsulation.

**Encapsulation and data hiding**

Encapsulation and data hiding are two fundamental concepts in object-oriented programming (OOP) that help improve the design, security, and maintainability of software. They are closely related but serve slightly different purposes.

1. **Encapsulation:**
   * **Definition:** Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit known as a class. It is one of the four fundamental OOP principles, the others being inheritance, polymorphism, and abstraction.
   * **Purpose:** Encapsulation helps in organizing code into logical units, making it easier to manage and maintain. It also enforces access controls on the data and methods, allowing for controlled and consistent interaction with the object.
2. **Data Hiding (Information Hiding):**
   * **Definition:** Data hiding is a specific aspect of encapsulation that refers to restricting the access to the internal state (data or attributes) of an object. It means that the data inside an object should not be directly accessible from outside the object.
   * **Purpose:** Data hiding provides a level of security and protection for an object's internal state. It prevents external code from directly modifying or accessing the object's data, ensuring that the object's internal integrity is maintained. This is achieved by using access modifiers like private, protected, and public in many programming languages.



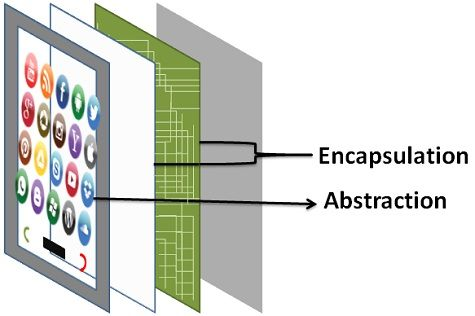
**Here's how encapsulation and data hiding work together:**

* A class encapsulates both its data (attributes) and the methods (functions) that operate on that data. These methods act as the interface through which external code interacts with the object.
* Data hiding is achieved by marking some of the class's attributes as private or protected, which means they can only be accessed or modified from within the class itself.
* To access or modify the encapsulated data, external code must use the public methods provided by the class. These methods serve as a controlled interface, allowing the class to enforce rules and validations on how data is accessed or modified.

**Encapsulation and abstraction**

Encapsulation and abstraction are two fundamental concepts in object-oriented programming (OOP) that help in designing and organizing code to create more manageable and maintainable software. They are closely related but serve different purposes:

1. **Encapsulation:**
   * Encapsulation is one of the four main principles of OOP, along with inheritance, polymorphism, and abstraction.
   * It refers to the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class.
   * The class serves as a blueprint or template for creating objects (instances) that have both data and behavior.

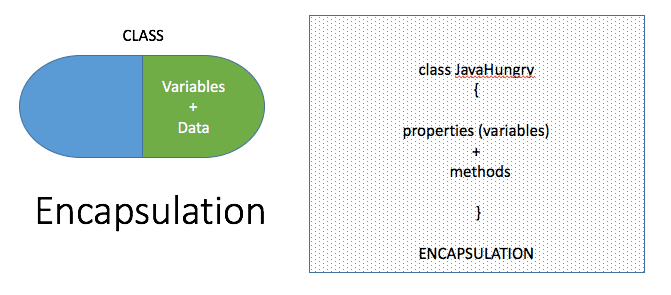


* + Encapsulation helps hide the internal implementation details of a class from the outside world and exposes only the necessary interfaces or methods to interact with the object's state.
  + Access to the internal data is controlled through access modifiers like public, private, and protected, ensuring that data is accessed and modified in a controlled manner.
  + Encapsulation promotes data integrity and reduces the risk of unintended data modification or misuse.

1. **Abstraction:**
   * Abstraction is the process of simplifying complex systems by breaking them down into more manageable parts while hiding unnecessary details.
   * In OOP, abstraction involves creating abstract classes or interfaces that define a set of methods or behaviors that subclasses must implement. These abstract classes or interfaces serve as blueprints for related classes.
   * Abstraction allows developers to focus on the essential characteristics of an object or system while ignoring the non-essential details.
   * It helps in managing complexity by providing a clear separation between what an object does (its behavior) and how it does it (its implementation).
   * Abstraction also enables code reusability through inheritance, as classes can inherit behavior from abstract classes or interfaces and then provide their own implementation of the required methods.

**Basic unit of encapsulation**

The basic unit of encapsulation in object-oriented programming (OOP) is the "class." Encapsulation is one of the four fundamental principles of OOP, and it refers to the bundling of data (attributes or properties) and the methods (functions or procedures) that operate on that data into a single unit called a class.



A class defines the blueprint or template for creating objects, and it encapsulates the data and behavior associated with those objects. The data members of a class are often declared as private or protected, meaning they can only be accessed and modified through the class's public methods or functions. This helps hide the internal implementation details of the class and provides a way to control access to the data, ensuring that it remains in a consistent and valid state.

**KT0303 Polymorphism**

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It is one of the four fundamental principles of OOP, along with encapsulation, inheritance, and abstraction.

Polymorphism enables you to write code that can work with objects of multiple types in a more generic and flexible way. There are two main types of polymorphism in OOP:

1. **Compile-time (or Static) Polymorphism**:
   * Compile-time polymorphism occurs when the decision about which method or function to call is made at compile time (i.e., before the program runs).
   * This is achieved through method overloading or operator overloading.
   * Method overloading allows you to define multiple methods in a class with the same name but different parameter lists. The appropriate method is selected at compile time based on the number or types of arguments passed.
   * Operator overloading allows you to define how operators (e.g., +, -, \*, /) behave when applied to objects of a class. The compiler selects the appropriate overloaded operator function at compile time based on the operands' types.
2. **Run-time (or Dynamic) Polymorphism:**
   * Run-time polymorphism occurs when the decision about which method or function to call is made at runtime (i.e., during program execution).
   * This is achieved through method overriding.
   * Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. When you call that method on an object of the subclass, the overridden method in the subclass is executed instead of the superclass's method. This is also known as "late binding" or "dynamic binding."

Polymorphism simplifies code by allowing you to write more generic and reusable functions and classes. It promotes flexibility and extensibility in your code because you can add new classes or types that inherit from a common superclass and still use them interchangeably with the existing code.

**Here's a simple example in Python to illustrate runtime polymorphism through method overriding:**

class Animal: def speak(self): pass class Dog(Animal): def speak(self): return "Woof!" class Cat(Animal): def speak(self): return "Meow!" def animal\_sound(animal): return animal.speak() dog = Dog() cat = Cat() print(animal\_sound(dog)) # Output: Woof! print(animal\_sound(cat)) # Output: Meow!

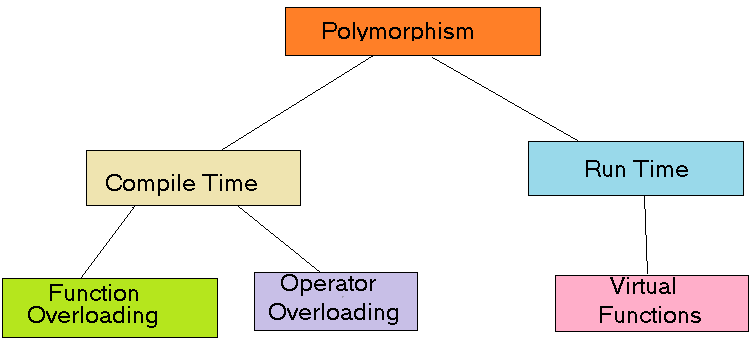
In this example, the animal\_sound function takes an Animal object as a parameter, and at runtime, it calls the appropriate speak method of the actual object (either a Dog or a Cat), demonstrating run-time polymorphism.

**Polymorphism and its types**

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables you to write code that can work with different types of objects in a uniform way, promoting code reusability and flexibility.

**There are two main types of polymorphism in OOP:**

1. Compile-Time Polymorphism (Static Binding or Early Binding):
   * Method Overloading: Method overloading allows a class to have multiple methods with the same name but different parameter lists. The compiler determines which method to invoke at compile time based on the number and types of arguments passed to the method.
   * Operator Overloading: Operator overloading allows you to define how operators (such as +, -, \*, /) behave for user-defined classes. This is done by defining special methods (e.g., operator+(), operator-()).



**Example of Method Overloading:**

class Calculator { int add(int a, int b) { return a + b; } double add(double a, double b) { return a + b; } }

1. Run-Time Polymorphism (Dynamic Binding or Late Binding):
   * Method Overriding: Method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The determination of which method to call happens at runtime, based on the actual object's type (dynamic type).

**Example of Method Overriding:**

class Shape { void draw() { System.out.println("Drawing a shape"); } } class Circle extends Shape { @Override void draw() { System.out.println("Drawing a circle"); } } public class Main { public static void main(String[] args) { Shape shape = new Circle(); // Upcasting shape.draw(); // Calls the overridden method in Circle } }

Polymorphism allows you to write more flexible and extensible code, as you can work with objects at a higher level of abstraction and take advantage of inheritance and method overriding to provide specific behaviors for different subclasses. It's a powerful concept that plays a central role in many object-oriented programming languages like Java, C++, and Python.



**Compile time polymorphism**

Compile-time polymorphism, also known as static polymorphism or early binding, is a type of polymorphism that is resolved during the compile-time phase of a program's execution. This means that the decision about which method or function to call is made by the compiler, not at runtime.

Compile-time polymorphism is achieved through method or function overloading and operator overloading. Here's a brief explanation of each:

1. Method Overloading: Method overloading occurs when a class has multiple methods with the same name but different parameters (either a different number of parameters or different types of parameters). The compiler determines which method to call based on the number and types of arguments passed during a function call.

class MathOperations { public int add(int a, int b) { return a + b; } public double add(double a, double b) { return a + b; } } public class Main { public static void main(String[] args) { MathOperations math = new MathOperations(); int sum1 = math.add(1, 2); // Calls the int version of add() double sum2 = math.add(1.5, 2.5); // Calls the double version of add() } }

In this example, the compiler knows which add method to call based on the argument types, and the decision is made at compile time.

1. Operator Overloading: Operator overloading involves defining multiple behaviors for the same operator depending on the operand’s types. Some programming languages, like C++, allow you to overload operators like +, -, \*, etc., for user-defined classes.

class Complex { public: int real, imag; Complex operator + (const Complex &other) { Complex result; result.real = this->real + other.real; result.imag = this->imag + other.imag; return result; } }; int main() { Complex a, b, c; a.real = 1; a.imag = 2; b.real = 3; b.imag = 4; c = a + b; // Calls the overloaded + operator return 0; }

In this C++ example, the + operator is overloaded for the Complex class, and the compiler knows which implementation of the + operator to use based on the operand’s types at compile time.

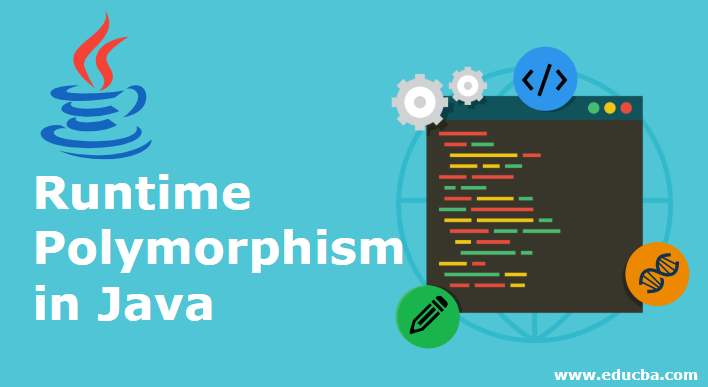
Compile-time polymorphism provides performance benefits because the compiler can optimize the code and eliminate runtime method resolution overhead. However, it comes with the limitation that the decision of which method or operator to call is made at compile time and cannot change dynamically at runtime, unlike runtime polymorphism (achieved through method overriding and virtual functions).

**Runtime polymorphism**

Runtime polymorphism, also known as dynamic polymorphism, is one of the fundamental concepts in object-oriented programming (OOP). It allows objects of different classes to be treated as objects of a common superclass during program execution. This enables you to write more flexible and extensible code because you can write code that can work with objects of different types without needing to know their specific types at compile-time.

Here's how runtime polymorphism works in most OOP languages like Java, C++, and Python:

1. **Inheritance:** First, you create a class hierarchy where one or more subclasses inherit from a common superclass. The superclass typically contains a set of methods or behaviors that are common to all its subclasses.
2. **Method Overriding:** In the subclasses, you can override (redefine) the methods inherited from the superclass. This means that each subclass can provide its own implementation of a method with the same name as that in the superclass.



1. **Polymorphic Reference**: You can create references to objects of the superclass type and use these references to refer to objects of the subclass type. This is possible because a subclass is considered to be a specialized version of the superclass.
2. **Dynamic Binding**: At runtime, the appropriate method implementation is determined based on the actual type of the object that the reference is pointing to, rather than the type of the reference itself. This is called dynamic binding or late binding.

**Here's a simple example in Java:**

class Animal { public void makeSound() { System.out.println("Some sound"); } } class Dog extends Animal { public void makeSound() { System.out.println("Woof!"); } } class Cat extends Animal { public void makeSound() { System.out.println("Meow!"); } } public class Main { public static void main(String[] args) { Animal myAnimal; myAnimal = new Dog(); myAnimal.makeSound(); // Calls Dog's makeSound() method myAnimal = new Cat(); myAnimal.makeSound(); // Calls Cat's makeSound() method } }

In this example, we have an Animal superclass with a makeSound method, and two subclasses, Dog and Cat, that override the makeSound method. At runtime, the makeSound method is dynamically bound to the appropriate implementation based on the actual type of the object (either Dog or Cat) that myAnimal is referring to.

Runtime polymorphism is a powerful concept in OOP that allows you to write more generic and flexible code, promoting code reusability and maintainability.

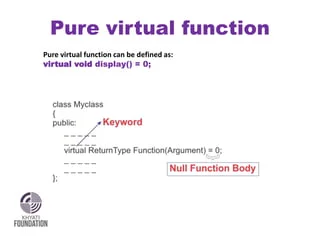
**Pure virtual function**

A pure virtual function in C++ is a special type of function that is declared in a base class (also known as an abstract base class) but does not have a definition in that base class. Instead, it is marked with the **= 0** syntax, which indicates that any derived class must provide an implementation for this function. Pure virtual functions are also sometimes referred to as abstract functions.

**Here's an example of how to declare a pure virtual function in C++:**

class Base { public: virtual void pureVirtualFunction() = 0; // Pure virtual function };

In this example, the pureVirtualFunction is declared as a pure virtual function in the Base class. This means that any class that derives from Base must provide an implementation for pureVirtualFunction. If a derived class fails to do so, it will also be considered an abstract class, and you won't be able to create objects of that class.



**Here's how you might implement this function in a derived class:**

class Derived : public Base { public: void pureVirtualFunction() override { // Provide an implementation for the pure virtual function // ... } };

By providing an implementation in the derived class, you make it a concrete class, and you can create objects of that class. Pure virtual functions are often used in C++ to define a common interface for a group of related classes while ensuring that each derived class implements the required functionality in its own way. This is a fundamental concept in object-oriented programming and is useful for achieving polymorphism and designing abstract base classes.

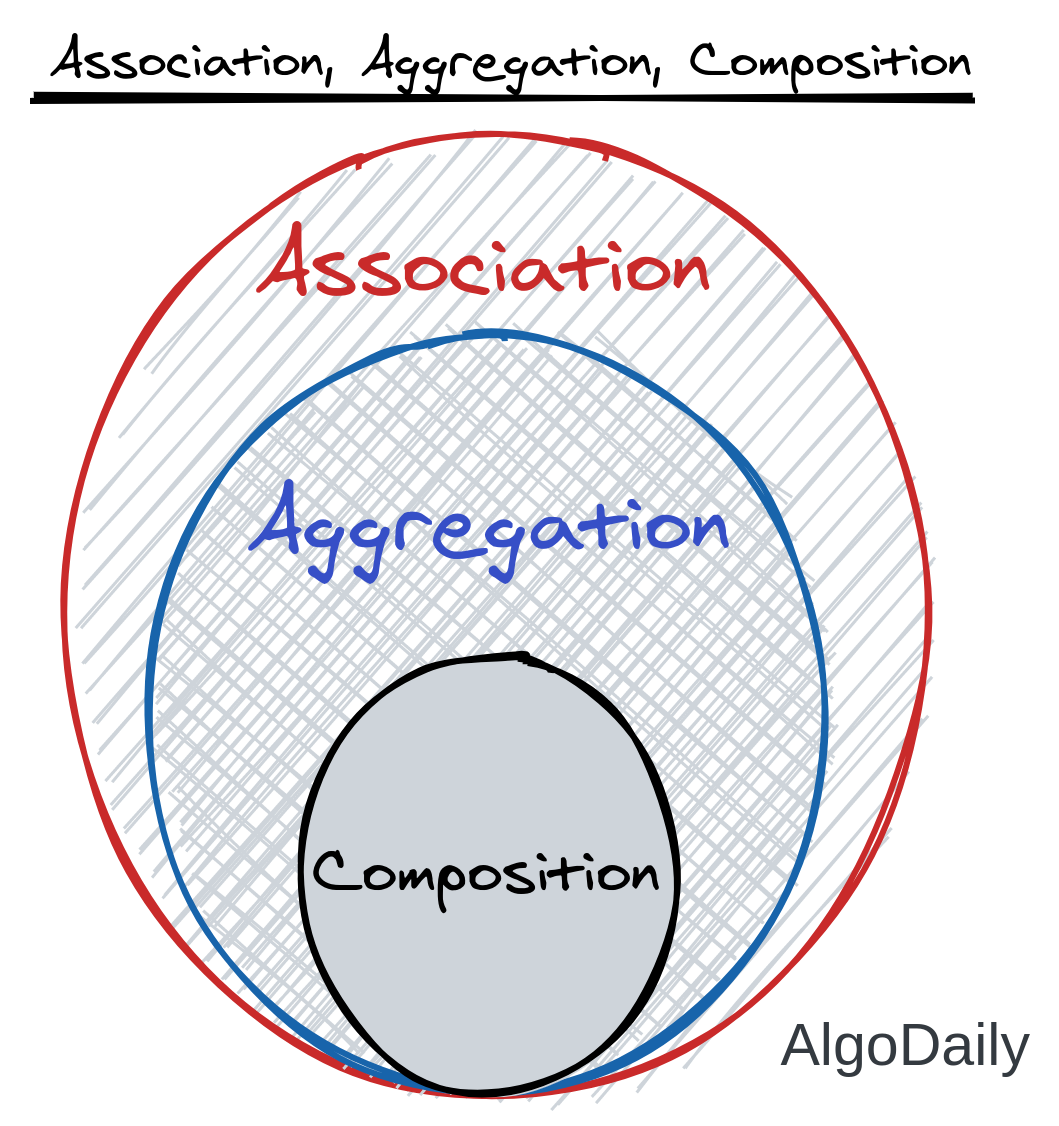
**KT0304 Association**

* **Organization or Group:** An association can refer to an organized group or body of people who have come together for a specific purpose or shared interest. Examples include trade associations, professional associations (like the American Medical Association or the American Bar Association), or community associations.
* **Mental Connection:** In psychology and cognitive science, association often refers to the mental process by which two or more concepts, ideas, or stimuli become linked or connected in a person's mind. This can involve concepts like classical conditioning (Pavlovian conditioning), where one stimulus becomes associated with another through repeated pairings.
* **Statistical Relationship:** In statistics and data analysis, association refers to the degree to which two or more variables are related or correlated. For example, if there is a positive association between the amount of exercise a person gets and their overall health, it means that as exercise increases, health tends to improve.
* **Legal or Business Term:** In legal and business contexts, association can refer to a formal relationship or partnership between individuals, companies, or entities. For example, two companies may form an association to collaborate on a project.
* **Memory and Recall:** In the context of memory and cognition, association can refer to the process by which one memory or piece of information triggers the recall of another related memory or piece of information. This is the basis of mnemonic devices and memory techniques.

**Aggregation and composition**

Aggregation and composition are two fundamental concepts in object-oriented programming (OOP) that describe relationships between classes and objects. They are used to model how objects interact with each other and how they can be organized to create more complex systems.

1. **Aggregation**: Aggregation represents a "has-a" or "part-of" relationship between objects. It is a way to design classes so that one class contains references to other classes as its members or attributes. The key characteristic of aggregation is that the child object (the one contained within another) can exist independently of the parent object. In other words, if the parent object is destroyed, the child object can still exist.



**Example:** In a university system, a department class can have an aggregation relationship with the Professor class. Each department has multiple professors, and professors can exist independently of the department. If the department is disbanded, the professors can still exist.

class Department {private List<Professor> professors; // Other department-related attributes and methods } class Professor { // Professor attributes and methods }

1. **Composition**: Composition is a stronger form of aggregation. It represents a "whole part" relationship, where one class (the whole) is composed of one or more other classes (the parts). In a composition relationship, the child objects are dependent on the parent object, and their lifecycles are tightly linked. If the parent object is destroyed, all its child objects are also typically destroyed.

Example: In a computer system, a Computer class can be composed of various components like a CPU, memory, and storage. If the computer is dismantled, its components are also typically removed.

class Computer { private CPU cpu; private Memory memory; private Storage storage; // Other computer-related attributes and methods } class CPU { // CPU attributes and methods } class Memory { // Memory attributes and methods } class Storage { // Storage attributes and methods }

In summary, aggregation and composition are used in object-oriented design to model relationships between classes and objects. Aggregation represents a weaker relationship where the child object can exist independently, while composition represents a stronger relationship where the child objects are tightly bound to the parent object's lifecycle. The choice between aggregation and composition depends on the specific requirements and design goals of the system you are building.

Top of Form

**HAS-A relationship between objects of two individual classes**

In object-oriented programming (OOP), the "HAS-A" relationship between objects of two individual classes is often referred to as composition or aggregation. This relationship represents that one class (the container or composite class) contains or is composed of objects of another class (the component class). It's a way to represent that an object of one class has a reference to an object of another class as one of its attributes or members.

Here's a brief explanation of composition and aggregation:

1. **Composition:**
   * In composition, the lifetime of the contained object is typically controlled by the containing object. This means that when the containing object is destroyed, the contained object is also destroyed.
   * Composition is often used when the contained object is an essential part of the containing object, and it cannot exist independently.
   * Composition is represented by a strong relationship. It's typically depicted by creating an instance of the component class within the container class and using it as an attribute.

**Example:**

class Engine: def start(self): print("Engine started") class Car: def \_\_init\_\_(self): self.engine = Engine() # Composition def start\_engine(self): self. engine.start() my\_car = Car() my\_car.start\_engine() # This also starts the engine object within the car.

1. **Aggregation:**
   * In aggregation, the lifetime of the contained object can be independent of the containing object. The contained object may exist before or after the containing object.
   * Aggregation is used when the relationship between the objects is more relaxed or when the contained object can exist on its own.
   * Aggregation is represented by a weaker relationship compared to composition.

**Example:**

class Department: def \_\_init\_\_(self, name): self.name = name class University: def \_\_init\_\_(self, name): self.name = name self.departments = [] # Aggregation def add\_department(self, department): self.departments.append(department) math\_dept = Department("Mathematics") physics\_dept = Department("Physics") my\_university = University("My University") my\_university.add\_department(math\_dept) my\_university.add\_department(physics\_dept) # The math\_dept and physics\_dept objects can exist independently of the my\_university object.

In summary, both composition and aggregation represent "HAS-A" relationships between objects of two individual classes, but they differ in terms of the ownership and lifetime management of the contained objects. Composition implies a stronger ownership relationship, while aggregation implies a weaker, more independent relationship.

**KT0305** **Benefits**

Benefits can refer to the advantages, rewards, or positive outcomes associated with various aspects of life, including work, health, education, and more. Here are some common categories of benefits and their associated advantages:

1. **Employee Benefits:**
   * **Healthcare Benefits:** These can include medical, dental, and vision insurance, which help employees and their families access necessary healthcare services.
   * **Retirement Benefits:** Such as 401(k) plans or pensions, provide financial security for employees in their later years.
   * **Paid Time Off (PTO):** Including vacation days, sick leave, and holidays, which allow employees to rest, recover, and spend time with loved ones.
   * **Flexible Work Arrangements:** Like remote work options or flexible hours, which improve work-life balance.
2. **Health Benefits:**
   * **Physical Health:** Regular exercise and a balanced diet can lead to improved physical fitness, reduced risk of chronic diseases, and increased longevity.
   * **Mental Health:** Practicing mindfulness, seeking therapy, or engaging in stress-reducing activities can improve mental well-being.
   * **Social Health:** Maintaining positive relationships with friends and family contributes to emotional well-being.
3. **Education Benefits:**
   * **Knowledge and Skills:** Education provides the knowledge and skills necessary to pursue career opportunities and personal growth.
   * **Career Advancement:** A higher level of education can open doors to more job prospects and higher earning potential.
4. **Financial Benefits:**
   * **Income:** Earning a salary or income provides financial security and the ability to meet basic needs.
   * **Savings and Investments:** Investing wisely and saving money can lead to financial stability and wealth accumulation.
5. **Social Benefits:**
   * **Community Engagement:** Involvement in community activities and volunteering can foster a sense of belonging and purpose.
   * **Social Connections:** Building and maintaining friendships and relationships can lead to emotional support and happiness.
6. **Environmental Benefits:**
   * **Environmental Conservation:** Sustainable practices and efforts to reduce pollution can lead to a healthier planet for current and future generations.
   * **Clean Energy:** Transitioning to clean energy sources can reduce greenhouse gas emissions and combat climate change.
7. **Technology Benefits:**
   * **Convenience:** Technology can make daily tasks easier and more efficient.
   * **Communication:** Tools like smartphones and social media enable instant communication and connection with others.
8. **Government Benefits:**
   * **Social Welfare Programs:** Government programs like unemployment benefits, food assistance, and housing subsidies can provide a safety net for those in need.
   * **Public Services:** Access to quality healthcare, education, and infrastructure services can improve overall quality of life.
9. **Business Benefits:**
   * **Profitability:** Effective business strategies can lead to increased revenue and profitability.
   * **Customer Satisfaction:** Providing value and quality products or services can lead to customer loyalty and repeat business.
10. **Personal Development Benefits:**
    * **Self-Improvement:** Engaging in lifelong learning, setting goals, and pursuing hobbies can lead to personal growth and fulfillment.
    * **Resilience:** Developing resilience and coping skills can help individuals navigate life's challenges effectively.

**Internal Assessment Criteria and Weight**

* 1. IAC0301 Capabilities of Object-Oriented Programming (OOP) in C++ are understood

**References**

1. *Yaiser, Michelle.*[*"Object-oriented programming concepts: Composition and aggregation"*](https://web.archive.org/web/20150408185639/http:/www.adobe.com/devnet/actionscript/learning/oop-concepts/composition-and-aggregation.html)*. Archived from*[*the original*](http://www.adobe.com/devnet/actionscript/learning/oop-concepts/composition-and-aggregation.html)*on April 8, 2015. There is a closely related concept to composition called aggregation. In conversation the differences between composition and aggregation are often ignored.*
2. [**^**](https://en.wikipedia.org/wiki/Object_composition#cite_ref-2) [*Design patterns : elements of reusable object-oriented software*](https://www.worldcat.org/oclc/31171684)*. Gamma, Erich., Helm, Richard (Computer scientist), Johnson, Ralph E., 1955-, Vlissides, John. Reading, Mass.: Addison-Wesley. 1995.*[*ISBN*](https://en.wikipedia.org/wiki/ISBN_(identifier))[*0-201-63361-2*](https://en.wikipedia.org/wiki/Special:BookSources/0-201-63361-2)*.*[*OCLC*](https://en.wikipedia.org/wiki/OCLC_(identifier))[*31171684*](https://www.worldcat.org/oclc/31171684)*.*
3. [**^**](https://en.wikipedia.org/wiki/Object_composition#cite_ref-3) *Ostermann, Klaus; Mezini, Mira (October 1, 2001).*[*"Object-oriented composition untangled"*](https://doi.org/10.1145/504311.504303)*. ACM SIGPLAN Notices.****36****(11): 283–299.*[*doi*](https://en.wikipedia.org/wiki/Doi_(identifier))*:*[*10.1145/504311.504303*](https://doi.org/10.1145%2F504311.504303)*.*[*ISSN*](https://en.wikipedia.org/wiki/ISSN_(identifier))[*0362-1340*](https://www.worldcat.org/issn/0362-1340)*.*
4. ^ [Jump up to:***a***](https://en.wikipedia.org/wiki/Object_composition#cite_ref-:0_4-0) [***b***](https://en.wikipedia.org/wiki/Object_composition#cite_ref-:0_4-1) *OMG (2017).*[*"Unified Modeling Language Specification Version 2.5.1"*](https://www.omg.org/spec/UML/About-UML/)*. www.omg.org. p. 109-110,197-201. Retrieved October 4, 2020.*
5. [**^**](https://en.wikipedia.org/wiki/Object_composition#cite_ref-5) [*"Aggregation"*](http://msdn2.microsoft.com/en-us/library/ms686558.aspx)*. Platform SDK for Windows XP SP2. Microsoft. Retrieved November 4, 2007.*
6. [**^**](https://en.wikipedia.org/wiki/Object_composition#cite_ref-6) *Sebesta, Robert W.*[*Concepts of Programming Languages*](https://archive.org/details/conceptsofprogra00sebe/page/218)*(Third ed.). Addison-Wesley Publishing Company, Inc. p.*[*218*](https://archive.org/details/conceptsofprogra00sebe/page/218)*.*[*ISBN*](https://en.wikipedia.org/wiki/ISBN_(identifier))[*0-8053-7133-8*](https://en.wikipedia.org/wiki/Special:BookSources/0-8053-7133-8)*.*