**900102-000-00-KM-03, Principles of Object-Oriented Programming with Java, NQF Level 4, Credits 4**

**Learner Guide**

**Module Three (3)**

|  |  |
| --- | --- |
| **Module Code** | 900102-000-00-KM-03 |
| **NQF Level** | 4 |
| **Credits** | 4 |
| **Skills Programme ID Number** | SP- 220329 |
| **Curriculum Title** | Java Programmer |
| **Curriculum Code** | 900102-000-00-00 |

|  |  |
| --- | --- |
| **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 fundamentals of Object-Oriented Programming with Java.

The learning will enable learners to demonstrate an understanding of:

* KM-03-KT01: Object-Oriented Programming System (OOPS) in Java 30%
* KM-03-KT02: Object-Oriented Programming in Java concepts 70%

**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:**

* Qualification of lecturer (SME):
* 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

**Exemptions**

* RPL based

**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**

A Java 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 Java 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 Java programs, using a disciplined coding style, including documentation and indentation standards.
* Use Git functionalities for working collaboratively in a team and execute version control.

**Skills Programme Rationale**

Realising the importance and future impact of the Fourth Industrial Revolution (4IR) on the economy of South Africa and its competitiveness, the Minister of Communications gazetted the Presidential Commission on the Fourth Industrial Revolution (PC4IR) on 9 April 2019. By March 2020 this Commission delivered a report with wide ranging recommendations for Human Capital Development that will drive the 4IR forward. It 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.

Software development is central to these initiatives. Software developers are the creative minds behind computer programs. Some develop the applications that allow people to do specific tasks on a computer or another device. Others develop the underlying systems that run the devices or that control networks. The software developer is the important cog in designing advanced computerised technologies. South Africa has a scarcity of software developers and there is a clear need for a qualification focusing specifically on the training and education of software developers.

**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 Java Programming

Associated Assessment Criteria (AACs)

* The fundamentals of the Java programming language are explained.
* The basic concepts and methods of object-oriented programming and object-oriented design are described.
* The development life-cycle as a means of creating applications is described.

**Exit Level Outcomes (ELO) 2**

Programme effectively using Java frameworks and functionalities

Associated Assessment Criteria (AACs)

* Java syntax is demonstrated, using the Java API.
* Well-written and readable Java programs are created, using a disciplined coding style, including documentation and indentation standards.
* Problems with application development are addressed by troubleshooting.

**Exit Level Outcomes (ELO) 3**

Work collaboratively in a team using GitHub platform

Associated Assessment Criteria (AACs)

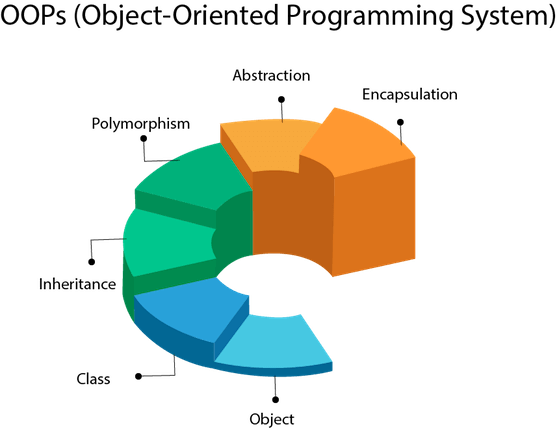
* An ability to work with GitHub is demonstrated.
* Working in a team collaboratively is achieved by using GitHub.
* Version control is exercised using GitHub. functionalities such as repositories, branches, commits and pull requests

**Session 1:** **KM-03-KT01: Object-Oriented Programming System (OOPS) in Java**

Topic elements to be covered include:

* KT0101 Definition
* KT0102 Concepts:
  + - Class and types
    - Object
    - Method and Method passing
* KT0103 Pillars/elements:
  + - Inheritance
    - Abstraction
    - Encapsulation
    - Polymorphism
    - Aggregation
    - Association
    - Composition
* KT0104 Advantages of OOPS

**KT0101 Definition**



Object-Oriented Programming (OOP) is a programming paradigm or approach that is used to design and implement software using objects, classes, and the principles of abstraction, encapsulation, inheritance, and polymorphism. In the context of Java, which is an object-oriented programming language, OOP refers to the system and principles used to structure and develop Java applications.

**Definition of Object-Oriented Programming System (OOPS) in Java:**

Object-Oriented Programming System (OOPS) in Java is a software development approach that organizes code into objects, where objects are instances of classes representing real-world entities and concepts. In OOP, Java applications are designed and constructed around the following key principles:

* **Classes and Objects:** Classes are blueprint templates that define the structure and behavior of objects. Objects are instances of classes and represent specific instances of data and functionality.
* **Abstraction:** Abstraction is the process of simplifying complex systems by modeling classes and objects that hide the unnecessary details while exposing essential characteristics.
* **Encapsulation:** Encapsulation is the practice of bundling data (attributes) and methods (functions) that operate on that data into a single unit called an object. Access to an object's internal state is controlled through access modifiers like public, private, and protected.
* **Inheritance:** Inheritance allows the creation of new classes (derived or child classes) based on existing classes (base or parent classes). It promotes code reuse and the extension of functionality.
* **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the flexibility to use different implementations interchangeably through method overriding and interfaces.

Java is a pure object-oriented language, meaning that everything in Java is treated as an object. It embodies OOP principles and provides features to create and manipulate objects effectively. Java also supports additional OOP concepts like interfaces and abstract classes, further enhancing its object-oriented capabilities.

OOPS in Java enables developers to write modular, maintainable, and scalable code by promoting the organization of code into reusable objects and classes. It encourages code flexibility, readability, and ease of maintenance, making it a powerful and widely used approach in software development.

**KT0102 Concepts:**

* **Class and types**
* **Object**
* **Method and Method passing**

In Object-Oriented Programming (OOP) in Java, several fundamental concepts play a crucial role in designing and structuring software applications. These concepts include classes and types, objects, methods, and method passing. Let's explore each of these concepts:

**1. Class and Types:**

* **Class:** A class is a blueprint or template that defines the structure and behavior of objects. It specifies the attributes (fields or properties) that an object of the class will have and the methods (functions) that it can perform. Classes are used to create objects.
* **Type:** In the context of Java, a type refers to the data type of a variable or an object. Each class defines a new data type, and objects of that class belong to that type. For example, if you have a class called **Car**, objects created from this class are of type **Car**.

**2. Object:**

* An object is an instance of a class. It represents a specific real-world entity or concept. Objects encapsulate data (attributes or properties) and behavior (methods or functions) that are defined in their respective classes. For example, if you have a **Car** class, an object of that class could represent a specific car with its unique attributes (e.g., make, model, year) and behavior (e.g., start, stop, accelerate).

**3. Method and Method Passing:**

* **Method:** A method is a block of code within a class that performs a specific task or operation. Methods define the behavior of objects created from the class. Methods can take parameters (input) and may return a value (output). In Java, methods are defined using the **public**, **private**, or **protected** access modifiers followed by a return type, a name, and a parameter list (if any).
* **Method Passing:** Method passing, also known as method invocation or method calling, refers to the process of invoking a method on an object. When a method is called on an object, it performs a specific action or computation using the object's data (attributes). Method passing can involve passing arguments to the method if it accepts parameters. For example:

// Method definition public void startEngine() { // Method body: Perform actions to start the car's engine } // Method invocation (method passing) Car myCar = new Car(); // Create a Car object myCar.startEngine(); // Call the startEngine() method on the Car object

In the example above, **startEngine()** is a method defined in the **Car** class, and it is invoked on the **myCar** object.

These fundamental concepts of classes, objects, methods, and method passing form the core of OOP in Java. They allow developers to model real-world entities as objects, define their attributes and behavior, and create modular and reusable code structures. By encapsulating data and methods within classes and interacting with objects, Java applications become more organized and maintainable.

**KT0103 Pillars/elements:**

* **Inheritance**
* **Abstraction**
* **Encapsulation**
* **Polymorphism**
* **Aggregation**
* **Association**
* **Composition**



In Object-Oriented Programming (OOP) in Java, there are several fundamental pillars or elements that serve as the foundation for designing and structuring software. These pillars help in achieving modularity, reusability, and maintainability in software development. Let's explore each of these OOP pillars in Java:

**1. Inheritance:**

* **Definition:** Inheritance is a mechanism that allows a new class (derived or child class) to inherit properties and behaviors from an existing class (base or parent class). It promotes code reuse and the creation of a hierarchy of classes.
* **Example:** If you have a **Vehicle** class, you can create derived classes like **Car** and **Bike** that inherit attributes and methods from the **Vehicle** class while adding their unique characteristics.

**2. Abstraction:**

* **Definition:** Abstraction is the process of simplifying complex systems by modeling classes and objects that hide the unnecessary details while exposing essential characteristics. It focuses on defining the essential properties and behaviors of an object.
* **Example:** When designing a **Shape** class, you can abstract common properties like **area** and **perimeter** without specifying the specific shapes (e.g., circle, rectangle) until subclasses are created.

**3. Encapsulation:**

* **Definition:** Encapsulation is the practice of bundling data (attributes) and methods (functions) that operate on that data into a single unit called an object. Access to an object's internal state is controlled through access modifiers like **public**, **private**, and **protected**.
* **Example:** Using access modifiers to control access to an object's attributes, such as making certain attributes **private** and providing public methods (getters and setters) to access or modify them.

**4. Polymorphism:**

* **Definition:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the flexibility to use different implementations interchangeably through method overriding and interfaces.
* **Example:** Creating an interface **Drawable** with a method **draw()**, which is implemented by various classes (e.g., **Circle**, **Rectangle**) to provide their own implementations of drawing.

**5. Aggregation:**

* **Definition:** Aggregation represents a "has-a" relationship between objects. It is a way to model objects that are composed of other objects. An aggregate object can exist independently and has a reference to one or more objects it contains.
* **Example:** A **University** class can have an aggregation relationship with **Department** objects, where a university has multiple departments.

**6. Association:**

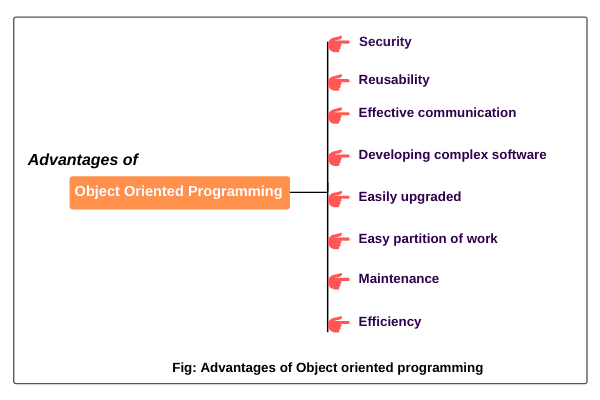
* **Definition:** Association represents a "uses" or "has-a" relationship between objects. It is a more general relationship than aggregation and can be used to model various interactions between objects.
* **Example:** A **Teacher** class and a **Student** class are associated with a **Course** class, representing the relationship between students, teachers, and the courses they take.

**7. Composition:**

* **Definition:** Composition is a stronger form of aggregation where the lifetime of the contained objects is tightly tied to the lifetime of the containing object. When the containing object is destroyed, the contained objects are also destroyed.
* **Example:** A **Library** class may compose **Book** objects, meaning that when the library is closed or destroyed, the books are also removed.

These pillars and elements of OOP in Java provide a rich set of concepts and tools for designing and building software applications. They enable developers to create modular, reusable, and maintainable code structures by modeling real-world entities and their relationships effectively. Understanding and applying these principles is fundamental to effective Java programming and software design.

**KT0104** **Advantages of OOPS**



Object-Oriented Programming (OOP) offers several advantages that make it a popular and widely used programming paradigm. These advantages contribute to improved code organization, maintainability, and reusability. In the context of Java and other object-oriented languages, here are some key advantages of OOP:

1. **Modularity:**
   * OOP promotes modularity by organizing code into self-contained objects and classes. Each object encapsulates both data (attributes) and behavior (methods), making it a modular unit that can be easily reused and maintained.
2. **Reusability:**
   * OOP allows for the creation of reusable classes and objects. Once a class is defined, it can be instantiated multiple times to create objects that inherit the same structure and behavior. This promotes code reuse, reducing development time and effort.
3. **Encapsulation:**
   * Encapsulation in OOP restricts direct access to an object's internal state (attributes) and allows controlled access through methods. This safeguards the integrity of the data and prevents unintended modifications, leading to more robust and secure code.
4. **Abstraction:**
   * Abstraction simplifies complex systems by modeling classes and objects that hide unnecessary implementation details. This concept enhances code clarity and reduces complexity, making it easier to understand and maintain.
5. **Inheritance:**
   * Inheritance allows the creation of new classes (derived or child classes) based on existing classes (base or parent classes). This promotes code reuse and the extension of functionality, leading to a more organized and efficient codebase.
6. **Polymorphism:**
   * Polymorphism enables the flexibility to use different implementations interchangeably. It allows objects of different classes to be treated as objects of a common superclass, simplifying code and enhancing adaptability.
7. **Easier Maintenance:**
   * OOP facilitates easier maintenance and updates to software systems. Changes can be made to individual classes or objects without affecting the entire codebase, reducing the risk of introducing new bugs.
8. **Better Collaboration:**
   * OOP encourages collaboration among developers by providing a structured and standardized way to design and implement software. Developers can work on individual classes or components, which can later be integrated into the overall system.
9. **Real-World Modeling:**
   * OOP closely aligns with real-world entities and concepts, making it an intuitive approach for modeling and solving real-world problems. This alignment enhances communication between developers and domain experts.
10. **Scalability:**
    * OOP supports the creation of scalable and extensible software systems. New classes and objects can be added to accommodate evolving requirements without disrupting existing functionality.
11. **Code Organization:**
    * OOP promotes a clean and organized code structure. Classes, objects, and their relationships are well-defined, making it easier to navigate and understand large codebases.
12. **Security:**
    * Encapsulation in OOP enhances security by controlling access to sensitive data and allowing for validation and access restrictions through methods.
13. **Testing and Debugging:**
    * OOP facilitates unit testing by isolating components (classes and objects) for testing. This makes it easier to identify and fix issues during development and maintenance.
14. **Community and Libraries:**
    * OOP languages like Java have extensive libraries and frameworks developed by the community, making it easier to leverage existing solutions for various tasks.

Overall, OOP provides a structured and efficient approach to software development, enhancing code quality, maintainability, and collaboration. These advantages make OOP a valuable paradigm for building complex and scalable software systems in Java and other object-oriented languages.

**Internal Assessment Criteria and Weight**

1. IAC0101 Definitions, functions and features of Object-Oriented Programming System (OOPS) in Java are stated

**Session 2:** **KM-03-KT02: Object-Oriented Programming in Java concepts 70%**

Topic elements to be covered include:

* KT0201 Basics of Designing a Class - Class, Object, State and Behaviour
* OOP Example - Fan Class - Deciding State and Constructors (exercise)
* OOP Example - Fan Class - Deciding Behaviour with Methods
* KT0202 Object Composition
  + - Customer Address Example
    - Books and Reviews
    - Exercise: Compose a class which addresses all the aspects of being human
* KT0203 Inheritance is a Method of Code Reuse
* KT0204 Class Hierarchies
* KT0205 Java Inheritance Basics
  + - What is Inherited?
    - Java Only Supports Singular Inheritance
* KT0206 Declaring Inheritance in Java
* KT0207 Inheritance and Type Casting
* Upcasting and Down casting
* KT0208 Overriding Methods
  + - The @override Annotation
    - Calling Superclass Methods
* KT0209 The instance of Instruction
* KT0210 Fields and Inheritance
* KT0211 Constructors and Inheritance
* KT0212 Nested Classes and Inheritance
* KT0213 Final Classes and Inheritance
* KT0214 Abstract Classes and Inheritance
* KT0215 Java Abstract Class
  + Creating Recipes with Template M
* KT0216 Java Interface
  + - How to think about interface
    - Complex Algorithm - API defined by extern
    - Unimplemented methods
* KT0217 Java Interface vs Abstract Class - A Comparison
* KT0218 Polymorphism
* KT0219 Introduction to abstraction
* KT0220 Introduction to Java constructors
  + - Rules for creating a constructor
    - Constructor overload in Java
* KT0221 Constructor chaining
* KT0222Understanding basics of Encapsulation with Setter methods
  + - Getters and generating getters and setters with <E>
* KT0223Initialisation of member variables
* KT0224First advantage of encapsulation

**KT0201 Basics of Designing a Class - Class, Object, State and Behaviour**

* **OOP Example - Fan Class - Deciding State and Constructors (exercise)**
* **OOP Example - Fan Class - Deciding Behaviour with Methods**



**1. Designing the Fan Class - State and Constructors:**

We'll start by defining the state (attributes) and constructors for the **Fan** class. In this example, a **Fan** object should have attributes to represent its state, such as **speed**, **isOn** (whether it's turned on or off), and **radius** (the size of the fan blades). We'll also create constructors to initialize these attributes.

public class Fan { // State attributes private int speed; // Speed of the fan (0, 1, 2) private boolean isOn; // Whether the fan is turned on (true or false) private double radius; // Radius of the fan blades // Constructors public Fan() { // Default constructor - initialize with default values speed = 0; // Default speed is 0 (off) isOn = false; // Default state is off radius = 5.0; // Default radius is 5.0 units } public Fan(int speed, boolean isOn, double radius) { // Parameterized constructor - initialize with specified values this.speed = speed; this.isOn = isOn; this.radius = radius; } // Additional methods for behavior will be defined later }

In the code above, we've created two constructors for the **Fan** class: a default constructor that initializes attributes with default values and a parameterized constructor that allows setting initial values.

**2. Defining Behavior with Methods:**

Next, let's define the behavior of the **Fan** class by adding methods. In this example, we'll create methods to turn the fan on or off, change its speed, and provide a method to get information about the fan's state.

public class Fan { // ... (previous code) // Method to turn on the fan public void turnOn() { isOn = true; speed = 1; // Default speed when turned on } // Method to turn off the fan public void turnOff() { isOn = false; speed = 0; // Fan is off, speed is 0 } // Method to increase the fan's speed public void increaseSpeed() { if (isOn && speed < 3) { speed++; } } // Method to decrease the fan's speed public void decreaseSpeed() { if (isOn && speed > 0) { speed--; } } // Method to get information about the fan public String getFanInfo() { String status = isOn ? "on" : "off"; return "The fan is " + status + " with speed " + speed + " and radius " + radius; } }

In this updated code, we've added methods such as **turnOn()**, **turnOff()**, **increaseSpeed()**, **decreaseSpeed()**, and **getFanInfo()** to control the fan's behavior and retrieve its state information.

Now, you can create **Fan** objects, set their state, and interact with them using these methods:

javaCopy code

Fan ceilingFan = new Fan(); // Create a fan object with default values ceilingFan.turnOn(); // Turn on the fan ceilingFan.increaseSpeed();// Increase fan speed System.out.println(ceilingFan.getFanInfo()); // Get fan information

This example demonstrates how to design a simple class, define its state and constructors, and implement its behavior using methods in an object-oriented manner.

**KT0202 Object Composition**

* **Customer Address Example**
* **Books and Reviews**
* **Exercise: Compose a class which addresses all the aspects of being human**

Object composition is a fundamental concept in object-oriented programming that involves creating complex objects by combining simpler objects. It allows you to build more intricate and meaningful structures by assembling various components. Let's explore object composition through examples:

**1. Customer Address Example:**

In this example, we'll create a **Customer** class that has a composition relationship with an **Address** class. The **Customer** class will have an **Address** object as one of its attributes.

public class Address { private String street; private String city; private String state; private String postalCode; // Constructor and methods for Address class... } public class Customer { private String firstName; private String lastName; private Address address; // Composition relationship with Address public Customer(String firstName, String lastName, Address address) { this.firstName = firstName; this.lastName = lastName; this.address = address; } // Methods and additional attributes for Customer class... }

In this composition example, a **Customer** object contains an **Address** object. The **Customer** class's constructor takes an **Address** object as a parameter, representing the composition relationship.

**2. Books and Reviews:**

Let's create a composition relationship between a **Book** class and a **Review** class. A **Book** can have multiple **Review** objects.

public class Review { private String reviewerName; private int rating; private String comment; // Constructor and methods for Review class... } public class Book { private String title; private String author; private List<Review> reviews; // Composition relationship with Review public Book(String title, String author) { this.title = title; this.author = author; this.reviews = new ArrayList<>(); } public void addReview(Review review) { reviews.add(review); } // Methods and additional attributes for Book class... }

In this composition example, a **Book** object contains a list of **Review** objects. The **addReview** method allows adding reviews to the book, representing the composition relationship.

**3. Exercise: Compose a class which addresses all aspects of being human:**

Let's compose a class called **Human** that encapsulates various aspects of being human, such as personal information, health, and skills. This class will demonstrate the composition of complex attributes within a single object.

public class Human { private String name; private int age; private Address address; private List<String> skills; private HealthStatus health; public Human(String name, int age, Address address, List<String> skills, HealthStatus health) { this.name = name; this.age = age; this.address = address; this.skills = skills; this.health = health; } // Methods and additional attributes for Human class... } public class Address { private String street; private String city; private String state; private String postalCode; // Constructor and methods for Address class... } public class HealthStatus { private double weight; private double height; private String bloodType; // Constructor and methods for HealthStatus class... }

In this composition exercise, the **Human** class is composed of an **Address** object for location information, a list of skills, and a **HealthStatus** object to represent health-related data.

Object composition allows you to create complex and meaningful objects by combining simpler objects, making it a powerful technique for modeling real-world relationships and structures in your Java programs.

**KT0203 Inheritance is a Method of Code Reuse**

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) in Java and serves as a method of code reuse and extending the functionality of existing classes. It allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). Inheritance promotes code reuse, modularity, and the creation of class hierarchies. Let's explore how inheritance works in Java:

**Key Concepts:**

1. **Base Class (Parent Class):**
   * A base class, also known as a parent class or superclass, is the existing class that provides the structure and behavior that can be inherited by other classes.
2. **Derived Class (Child Class):**
   * A derived class, also known as a child class or subclass, is a new class that is created by inheriting attributes and methods from a base class. The derived class can also add its own attributes and methods.

**Benefits of Inheritance:**

* **Code Reuse:** Inheritance allows you to reuse code from existing classes. Instead of writing similar code from scratch, you can create new classes that inherit attributes and methods from a base class.
* **Modularity:** Inheritance promotes the creation of modular and organized code structures. Each class focuses on a specific aspect of an object, making the code easier to understand and maintain.
* **Hierarchy:** You can create class hierarchies where a base class serves as a general template, and derived classes provide more specific implementations. This hierarchy mirrors real-world relationships and promotes a natural way of modeling objects.

**Syntax of Inheritance in Java:**

In Java, the **extends** keyword is used to establish an inheritance relationship between classes. Here's the basic syntax:

class ParentClass { // Attributes and methods of the parent class } class ChildClass extends ParentClass { // Additional attributes and methods of the child class }

**Example:**

Let's consider an example involving two classes: **Vehicle** (parent class) and **Car** (child class).

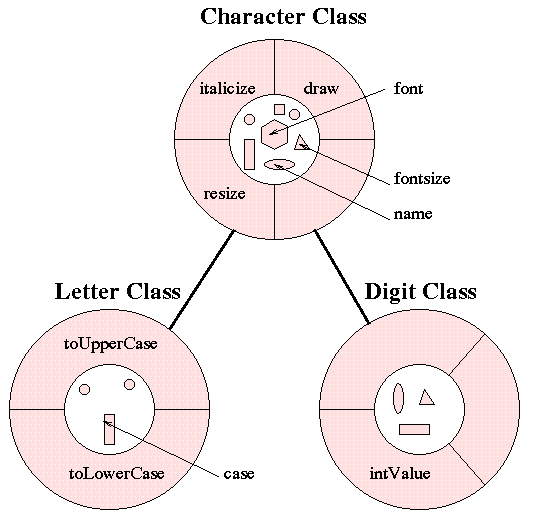
class Vehicle { String make; String model; public Vehicle(String make, String model) { this.make = make; this.model = model; } void start() { System.out.println("Starting the vehicle."); } } class Car extends Vehicle { int year; public Car(String make, String model, int year) { super(make, model); // Call the constructor of the parent class this.year = year; } void accelerate() { System.out.println("Accelerating the car."); } }

In this example, the **Car** class inherits the attributes **make** and **model** as well as the **start** method from the **Vehicle** class. It adds its own attribute **year** and method **accelerate**. This allows you to create **Car** objects that have both the characteristics of vehicles and car-specific features.

Car myCar = new Car("Toyota", "Camry", 2023); myCar.start(); // Inherited method from Vehicle myCar.accelerate(); // Car-specific method

Inheritance is a powerful mechanism in Java that enhances code organization, promotes code reuse, and enables the creation of complex class hierarchies to model real-world relationships effectively.

**KT0204 Class Hierarchies**



In Object-Oriented Programming (OOP) in Java, class hierarchies play a crucial role in modeling real-world relationships and organizing classes in a structured manner. A class hierarchy represents a hierarchical structure of classes where some classes (child or derived classes) inherit attributes and methods from other classes (parent or base classes). Class hierarchies are a fundamental concept for building complex and organized software systems. Let's delve into class hierarchies in Java:

**Key Concepts:**

1. **Base Class (Parent Class):**
   * A base class, also known as a parent class or superclass, is the top-level class in the hierarchy. It defines common attributes and methods that can be inherited by other classes.
2. **Derived Class (Child Class):**
   * A derived class, also known as a child class or subclass, is a class that inherits attributes and methods from a base class. It can also add its own attributes and methods.
3. **Inheritance Relationship:**
   * Inheritance establishes a relationship between classes in the hierarchy, where derived classes inherit properties (attributes and methods) from their parent classes.
4. **Overriding:**
   * In Java, derived classes can override (provide their own implementation of) methods inherited from their parent classes. This allows for customization of behavior in the derived class.

**Benefits of Class Hierarchies:**

* **Code Reuse:** Class hierarchies enable the reuse of code from base classes. Common attributes and methods need to be defined only once in the parent class and can be inherited by multiple child classes.
* **Organization:** Class hierarchies provide a structured way to organize classes. Related classes can be grouped together in a hierarchy, making the code more organized and easier to maintain.
* **Polymorphism:** Class hierarchies support polymorphism, allowing objects of different classes (child and parent) to be treated as objects of a common superclass. This promotes flexibility in coding and enables the use of interfaces and abstract classes.

**Example of Class Hierarchy:**

Consider a class hierarchy involving animals, where we have a base class **Animal** and derived classes such as **Mammal**, **Bird**, and **Fish**. Each derived class inherits attributes and methods from the **Animal** class while adding its own characteristics.

class Animal { String name; int age; public Animal(String name, int age) { this.name = name; this.age = age; } void eat() { System.out.println(name + " is eating."); } } class Mammal extends Animal { String furColor; public Mammal(String name, int age, String furColor) { super(name, age); // Call the constructor of the parent class this.furColor = furColor; } void giveBirth() { System.out.println(name + " is giving birth."); } } class Bird extends Animal { String featherColor; public Bird(String name, int age, String featherColor) { super(name, age); this.featherColor = featherColor; } void fly() { System.out.println(name + " is flying."); } } class Fish extends Animal { String finType; public Fish(String name, int age, String finType) { super(name, age); this.finType = finType; } void swim() { System.out.println(name + " is swimming."); } }

In this hierarchy, the **Animal** class defines common attributes and methods for all animals, while the derived classes (**Mammal**, **Bird**, and **Fish**) inherit these properties and add their own specific attributes and methods.

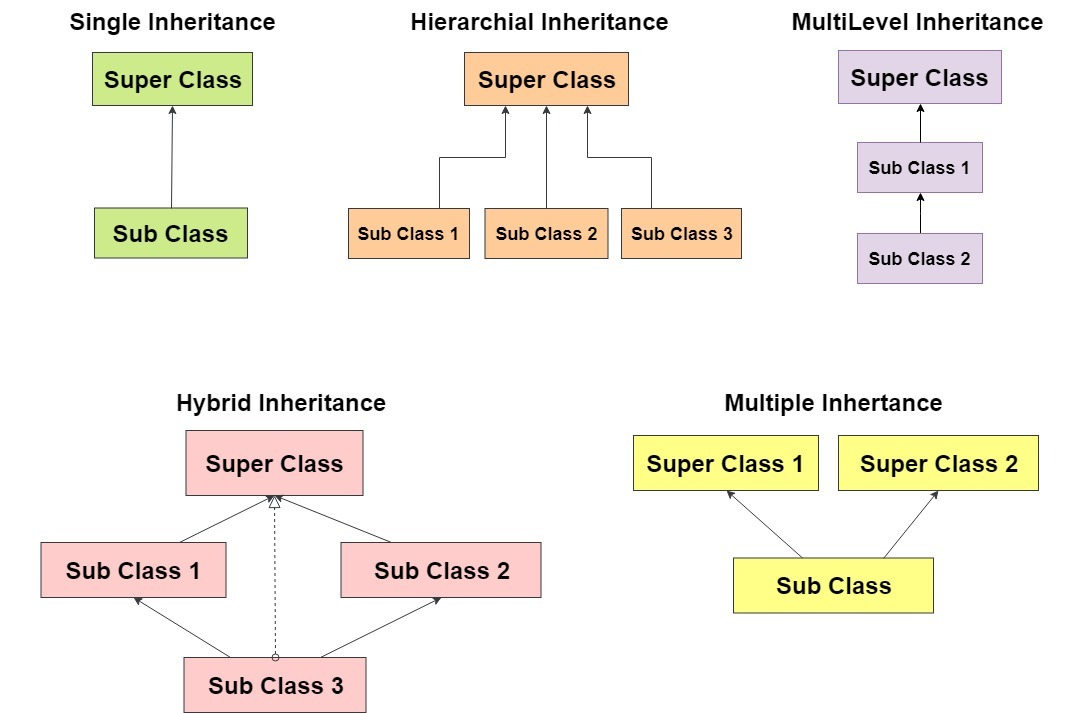
Mammal dog = new Mammal("Dog", 5, "Brown"); dog.eat(); // Inherited from Animal dog.giveBirth(); // Mammal-specific method Bird sparrow = new Bird("Sparrow", 2, "Gray"); sparrow.eat(); // Inherited from Animal sparrow.fly(); // Bird-specific method

Class hierarchies in Java help create organized, reusable, and extensible code structures, making it easier to model and work with complex systems and relationships. They are a fundamental concept in OOP and facilitate the creation of modular and maintainable software.

**KT0205 Java Inheritance Basics**

* **What is Inherited?**
* **Java Only Supports Singular Inheritance**

**Java Inheritance Basics:**



Inheritance is a fundamental concept in Java and object-oriented programming (OOP) that allows you to create new classes (child or derived classes) based on existing classes (parent or base classes). Inheritance enables code reuse and the creation of class hierarchies. Here are some key aspects of Java inheritance:

**1. What is Inherited?**

* Inheritance allows child classes to inherit attributes (fields) and methods from their parent classes. This means that child classes automatically have access to the fields and methods defined in the parent class.
* Inherited attributes and methods can be used directly in the child class without redefining them, promoting code reuse and modularity.
* Inheritance establishes an "is-a" relationship, where a child class "is a" type of its parent class.

**2. Java Only Supports Singular Inheritance:**

* Java supports single inheritance, which means that a class can inherit from only one parent class.
* In other words, a Java class can have only one direct superclass (parent class).
* This limitation is in contrast to some other programming languages that support multiple inheritance, where a class can inherit from multiple parent classes.

**Example of Inheritance in Java:**

class Vehicle { String make; String model; public Vehicle(String make, String model) { this.make = make; this.model = model; } void start() { System.out.println("Starting the vehicle."); } } class Car extends Vehicle { int year; public Car(String make, String model, int year) { super(make, model); // Call the constructor of the parent class this.year = year; } void accelerate() { System.out.println("Accelerating the car."); } }

In this example, the **Car** class inherits attributes and methods from the **Vehicle** class. The **Car** class can access the **make** and **model** attributes and the **start** method without redefining them.

Car myCar = new Car("Toyota", "Camry", 2023); myCar.start(); // Inherited method from Vehicle myCar.accelerate(); // Car-specific method

While Java supports single inheritance for classes, it allows multiple levels of inheritance, where a child class can itself become a parent class for another child class. This allows the creation of complex class hierarchies while adhering to the single inheritance constraint.

In summary, Java inheritance allows child classes to inherit attributes and methods from parent classes, facilitating code reuse and the creation of class hierarchies. Java supports single inheritance, meaning that a class can inherit from only one parent class, but it can have multiple levels of inheritance.

**KT0206 Declaring Inheritance in Java**

In Java, you declare inheritance using the **extends** keyword when defining a new class. This keyword establishes an inheritance relationship between the new class (the child or derived class) and an existing class (the parent or base class). Here's the syntax for declaring inheritance in Java:

class ChildClass extends ParentClass { // Class members (attributes and methods) for the child class }

Here's an explanation of each part of the syntax:

* **ChildClass**: This is the name of the new class you're creating, which will inherit attributes and methods from the parent class.
* **extends**: This keyword indicates that **ChildClass** is inheriting from **ParentClass**.
* **ParentClass**: This is the name of the existing class (the parent class) from which **ChildClass** will inherit attributes and methods.

Once you declare this inheritance relationship, **ChildClass** will have access to all the non-private attributes and methods of **ParentClass**. You can use them directly in **ChildClass** without redefining them, promoting code reuse.

Here's a simple example:

class Animal { void eat() { System.out.println("Animal is eating."); } } class Dog extends Animal { void bark() { System.out.println("Dog is barking."); } }

In this example, **Dog** is declared to extend **Animal**, indicating that **Dog** inherits from **Animal**. Therefore, **Dog** has access to the **eat** method from **Animal** and can also define its own method **bark**. You can create **Dog** objects and call both **eat** and **bark** methods on them.

Dog myDog = new Dog(); myDog.eat(); // Inherited method from Animal myDog.bark(); // Dog-specific method

This demonstrates how to declare and use inheritance in Java to create class hierarchies and promote code reuse.

**KT0207 Inheritance and Type Casting**

Inheritance and type casting are two important concepts in Object-Oriented Programming (OOP) in Java that are often used together when dealing with class hierarchies. Let's explore how inheritance and type casting work together:

**1. Inheritance:**

* Inheritance is a mechanism in Java where a new class (child or derived class) can inherit attributes and methods from an existing class (parent or base class). This allows for code reuse and the creation of class hierarchies.

**2. Type Casting:**

* Type casting is the process of converting an object from one data type to another. In the context of inheritance, it is commonly used to treat an object of a derived class as an object of its parent class or vice versa.

**Using Type Casting in Inheritance:**

When you have a class hierarchy with inheritance, you can use type casting to:

**a. Upcasting (Implicit Casting):**

* Upcasting is the process of treating an object of a derived class as an object of its parent class. This is also known as widening or implicit casting and doesn't require an explicit cast operator.

**b. Downcasting (Explicit Casting):**

* Downcasting is the process of treating an object of a parent class as an object of its derived class. This is also known as narrowing or explicit casting and requires the use of an explicit cast operator.

**Example of Upcasting (Implicit Casting):**

class Animal { void eat() { System.out.println("Animal is eating."); } } class Dog extends Animal { void bark() { System.out.println("Dog is barking."); } } public class Main { public static void main(String[] args) { Animal myDog = new Dog(); // Upcasting myDog.eat(); // Accessing the eat() method of Animal // myDog.bark(); // This would result in a compilation error because the reference is of type Animal } }

In this example, **myDog** is upcasted to an **Animal** reference, allowing us to access the **eat** method from the parent class **Animal**. However, we cannot access the **bark** method of the **Dog** class using the **myDog** reference because it's treated as an **Animal**.

**Example of Downcasting (Explicit Casting):**

class Animal { void eat() { System.out.println("Animal is eating."); } } class Dog extends Animal { void bark() { System.out.println("Dog is barking."); } } public class Main { public static void main(String[] args) { Animal myAnimal = new Dog(); // Upcasting if (myAnimal instanceof Dog) { Dog myDog = (Dog) myAnimal; // Downcasting (Explicit Casting) myDog.bark(); // Accessing the bark() method of Dog } } }

In this example, we first upcast **myAnimal** to an **Animal** reference. Then, using the **instanceof** operator, we check if it's an instance of **Dog**. If it is, we perform explicit downcasting to access the **bark** method of the **Dog** class.

Using inheritance and type casting together allows you to work with class hierarchies effectively, switch between parent and child classes as needed, and access specific behaviors or attributes of derived classes when required. However, it's important to use type casting carefully to avoid runtime errors.

* **Upcasting and Down casting**

**Upcasting and Downcasting** are two fundamental concepts in object-oriented programming, particularly in languages like Java. They relate to how you work with class hierarchies, inheritance, and type casting. Let's explore both of these concepts:

**Upcasting (Widening or Implicit Casting):**

* Upcasting is the process of converting an object from a derived class to a parent class. It allows you to treat an object of a derived class as an object of its parent class.
* Upcasting is implicit, which means you don't need to use an explicit cast operator. It's done automatically by the compiler.
* Upcasting is safe and always allowed because a derived class object can be used wherever a parent class object is expected.
* The primary purpose of upcasting is to achieve polymorphism, where objects of different classes can be treated uniformly.

Here's an example in Java:

class Animal { void eat() { System.out.println("Animal is eating."); } } class Dog extends Animal { void bark() { System.out.println("Dog is barking."); } } public class Main { public static void main(String[] args) { Animal myAnimal = new Dog(); // Upcasting myAnimal.eat(); // Accessing the eat() method of Animal // myAnimal.bark(); // Compilation error because myAnimal is treated as an Animal } }

In this example, **myAnimal** is upcasted to an **Animal** reference, allowing us to access the **eat** method from the parent class **Animal**. However, we cannot access the **bark** method of the **Dog** class using the **myAnimal** reference because it's treated as an **Animal**.

**Downcasting (Narrowing or Explicit Casting):**

* Downcasting is the process of converting an object from a parent class to a derived class. It allows you to treat an object of a parent class as an object of its derived class.
* Downcasting is explicit and requires using an explicit cast operator.
* Downcasting can be risky because if the object being downcasted is not an instance of the derived class, it will result in a runtime **ClassCastException**.
* You should use the **instanceof** operator to check the object's type before performing downcasting to avoid runtime errors.

Here's an example in Java:

class Animal { void eat() { System.out.println("Animal is eating."); } } class Dog extends Animal { void bark() { System.out.println("Dog is barking."); } } public class Main { public static void main(String[] args) { Animal myAnimal = new Dog(); // Upcasting if (myAnimal instanceof Dog) { Dog myDog = (Dog) myAnimal; // Downcasting (Explicit Casting) myDog.bark(); // Accessing the bark() method of Dog } } }

In this example, we first upcast **myAnimal** to an **Animal** reference and then check whether it's an instance of **Dog** using the **instanceof** operator. If it is, we perform explicit downcasting to access the **bark** method of the **Dog** class.

In summary, upcasting and downcasting are essential techniques in working with class hierarchies and achieving polymorphism in object-oriented programming. Upcasting is implicit and safe, while downcasting is explicit and requires caution to avoid runtime errors.

**KT0208 Overriding Methods**

* **The @override Annotation**
* **Calling Superclass Methods**

**Overriding Methods** is a fundamental concept in Object-Oriented Programming (OOP) that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. When a method in a subclass has the same name, return type, and parameters as a method in its superclass, it is said to override that method. Here's more information on overriding methods in Java:

**1. The @Override Annotation:**

* In Java, you can use the **@Override** annotation before a method in a subclass to indicate that you intend to override a method from the superclass. This annotation helps catch potential errors during compilation by ensuring that you're correctly overriding a method.
* If the method in the subclass doesn't match the signature of any method in the superclass, or if there's no corresponding method in the superclass, a compilation error will occur.
* The **@Override** annotation is optional but considered good practice when overriding methods.

**Example of Overriding Methods:**

class Animal { void makeSound() { System.out.println("Animal makes a sound."); } } class Dog extends Animal { @Override // Optional annotation indicating method overriding void makeSound() { System.out.println("Dog barks."); } }

In this example, the **Dog** class overrides the **makeSound** method of the **Animal** class. The **@Override** annotation is used to indicate that the method in **Dog** is intended to override the method in **Animal**.

**2. Calling Superclass Methods:**

* When you override a method in a subclass, you may want to call the superclass's version of the method within the overridden method. This is achieved using the **super** keyword.
* The **super** keyword allows you to explicitly call a method from the superclass with the same name.
* This is often done to reuse some of the functionality from the superclass and then add or modify behavior in the subclass.

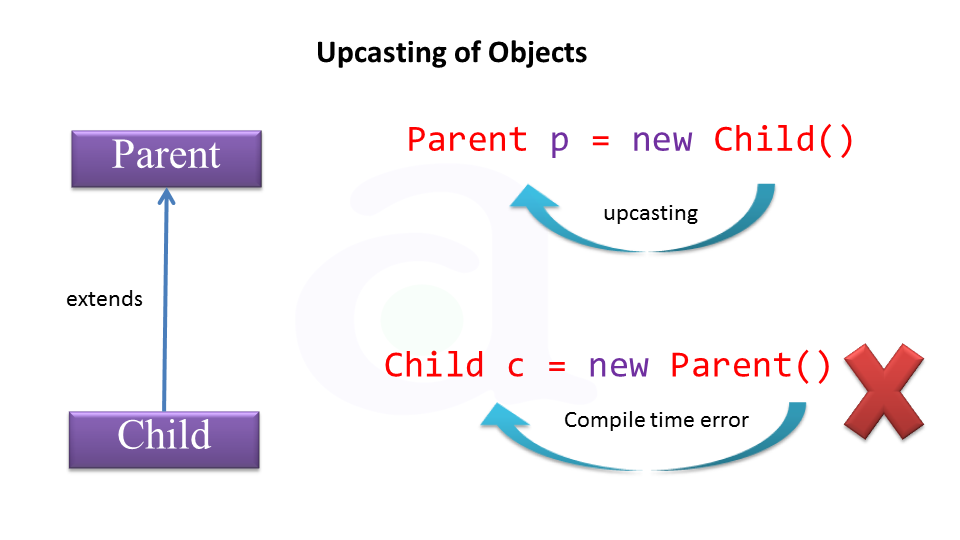
**Example of Calling Superclass Methods:**

class Animal { void makeSound() { System.out.println("Animal makes a sound."); } } class Dog extends Animal { @Override void makeSound() { super.makeSound(); // Calling the superclass's makeSound method System.out.println("Dog barks."); } }

In this example, within the **makeSound** method of the **Dog** class, **super.makeSound()** is used to call the **makeSound** method from the **Animal** superclass. This allows the **Dog** class to reuse the behavior from the superclass and then add its own behavior (printing "Dog barks.").

Overriding methods and using the **@Override** annotation, along with calling superclass methods using **super**, are important techniques in OOP for achieving polymorphism and customizing behavior in subclasses while maintaining a consistent interface defined by the superclass.

**KT0209 The instance of Instruction**



The **instanceof** operator is a fundamental concept in Java and is used to test whether an object is an instance of a particular class, interface, or subclass. It's particularly useful when working with inheritance, polymorphism, and type checking. Here's how the **instanceof** operator works:

**Syntax:**

boolean result = object instanceof ClassType;

* **object**: This is the object you want to check.
* **ClassType**: This is the class or interface you want to check if the object is an instance of.

**Usage:**

* The **instanceof** operator returns **true** if the **object** is an instance of **ClassType** or a subclass of **ClassType**.
* It returns **false** if the **object** is not an instance of **ClassType**.

**Example:**

class Animal { // Animal class definition } class Dog extends Animal { // Dog class definition } public class Main { public static void main(String[] args) { Animal myAnimal = new Dog(); if (myAnimal instanceof Animal) { System.out.println("myAnimal is an instance of Animal"); } if (myAnimal instanceof Dog) { System.out.println("myAnimal is an instance of Dog"); } } }

In this example, **myAnimal** is an instance of the **Dog** class, which is a subclass of **Animal**. Therefore, both **instanceof** checks return **true**, indicating that **myAnimal** is an instance of both **Animal** and **Dog**.

**Common Use Cases:**

* Checking the type of an object before performing a type-specific operation to avoid runtime errors.
* Implementing custom logic based on the specific subclass of an object.
* Working with interfaces to check if an object implements a particular interface.

**Note:**

* While **instanceof** is a useful tool, it's often recommended to use polymorphism and inheritance effectively instead of relying too heavily on **instanceof** checks. In many cases, good design practices can help you avoid the need for extensive type checking.

**KT0210 Fields and Inheritance**

Fields and inheritance are closely related concepts in object-oriented programming (OOP), especially in languages like Java. Let's explore how fields (also known as attributes or member variables) and inheritance interact:

**1. Inherited Fields:**

* Inheritance allows a subclass (child class) to inherit fields from its superclass (parent class).
* Fields declared in the parent class are automatically accessible in the child class, assuming they have appropriate access modifiers (e.g., **public**, **protected**, or **package-private**). Private fields are not directly accessible in child classes.
* Inherited fields maintain their original data types, names, and any access modifiers from the parent class.

**Example of Inherited Fields:**

class Vehicle { String make; String model; protected int year; public Vehicle(String make, String model, int year) { this.make = make; this.model = model; this.year = year; } } class Car extends Vehicle { // Inherited fields make, model, and year private boolean automatic; public Car(String make, String model, int year, boolean automatic) { super(make, model, year); // Call the parent class constructor this.automatic = automatic; } public void displayInfo() { System.out.println("Make: " + make); System.out.println("Model: " + model); System.out.println("Year: " + year); System.out.println("Automatic Transmission: " + automatic); } }

In this example, the **Car** class inherits the fields **make**, **model**, and **year** from the **Vehicle** class. The **protected** access modifier is used for the **year** field, allowing it to be accessed within the **Car** class.

**2. Hiding Fields:**

* If a subclass declares a field with the same name as a field in its superclass, it hides the inherited field. This is known as field hiding.
* When you access the field in the subclass, the field declared in the subclass takes precedence over the inherited field.
* You can access the inherited field using the **super** keyword.

**Example of Field Hiding:**

class Vehicle { String make; String model; public Vehicle(String make, String model) { this.make = make; this.model = model; } } class Car extends Vehicle { String model; // Hides the inherited model field public Car(String make, String model, String carModel) { super(make, model); // Call the parent class constructor this.model = carModel; // Hides the inherited model field } public void displayInfo() { System.out.println("Make: " + make); System.out.println("Car Model (subclass): " + model); // Accessing the subclass field System.out.println("Car Model (superclass): " + super.model); // Accessing the inherited field } }

In this example, the **Car** class declares a **model** field, which hides the inherited **model** field from the **Vehicle** class. The **super.model** expression is used to access the inherited field.

Fields and inheritance allow you to create class hierarchies and share data between classes in a structured way. Careful use of fields and consideration of access modifiers are important for proper encapsulation and data hiding in your OOP design.

**KT0211 Constructors and Inheritance**

Constructors and inheritance are closely connected concepts in object-oriented programming, particularly in languages like Java. Inheritance allows child classes to inherit attributes and behaviors from their parent classes, including constructors. Here's how constructors and inheritance work together:

**1. Inheriting Constructors:**

* When a child class is created, it can inherit constructors from its parent class. In other words, the child class can use the same constructors as the parent class.
* The child class can also define its own constructors in addition to inheriting the constructors from the parent class.
* Constructors are not directly inherited as methods or fields; instead, they are implicitly available to the child class.

**2. Constructor Chaining:**

* Constructor chaining occurs when a constructor in a child class explicitly calls a constructor from its parent class using the **super** keyword. This allows you to initialize the inherited attributes defined in the parent class.
* If a constructor in the child class doesn't explicitly call a parent class constructor using **super**, Java implicitly adds a call to the default (parameterless) constructor of the parent class.

**Example of Inherited Constructors:**

class Vehicle { String make; String model; public Vehicle(String make, String model) { this.make = make; this.model = model; } } class Car extends Vehicle { int year; public Car(String make, String model, int year) { super(make, model); // Call the parent class constructor this.year = year; } public void displayInfo() { System.out.println("Make: " + make); System.out.println("Model: " + model); System.out.println("Year: " + year); } }

In this example, the **Car** class inherits the constructor **Vehicle(String make, String model)** from the **Vehicle** class. The **Car** class defines its own constructor, **Car(String make, String model, int year)**, which explicitly calls the parent class constructor using **super(make, model)**. This allows the **Car** class to initialize its own attributes (**year**) and the inherited attributes (**make** and **model**) properly.

**3. Default Constructors:**

* If the parent class has no constructors explicitly defined, Java provides a default constructor with no arguments.
* If the parent class has one or more constructors explicitly defined, Java does not provide a default constructor. In such cases, if you want to create an instance of the child class without arguments, you need to provide your own default constructor in both the parent and child classes.

**Example with Default Constructors:**

class Vehicle { String make; String model; } class Car extends Vehicle { int year; public Car() { // No-argument constructor for Car this.make = "Unknown"; this.model = "Unknown"; this.year = 0; } public void displayInfo() { System.out.println("Make: " + make); System.out.println("Model: " + model); System.out.println("Year: " + year); } }

In this example, both the **Vehicle** and **Car** classes have a default constructor, allowing you to create instances of the **Car** class without specifying any arguments.

Constructors and inheritance work together to ensure that objects of child classes are properly initialized, and constructors from parent classes are used as needed. It's important to understand how constructors are inherited and how constructor chaining can be used to initialize objects in class hierarchies.

**KT0212 Nested Classes and Inheritance**

**Nested Classes and Inheritance in Java**

Nested classes, also known as inner classes, are classes defined within the scope of another class. They can be used in combination with inheritance, but there are some considerations to keep in mind when working with nested classes and inheritance in Java.

**1. Nested Classes in Java:**

* Java supports four types of nested classes: static nested classes, non-static nested classes (inner classes), local classes, and anonymous classes.
* The relationship between a nested class and its enclosing class is similar to that of a member variable or method; the nested class can access the members (variables and methods) of its enclosing class.

**2. Inheritance with Nested Classes:**

* Nested classes can participate in inheritance like any other class. They can be extended by other classes, and they can extend other classes, including other nested classes.
* When a nested class extends a class, it inherits the attributes and methods of the parent class, including those inherited from its own parent class (if any).
* When a nested class extends a non-static nested class (inner class), it also inherits a reference to the outer class (enclosing class) object, which can be accessed using **OuterClassName.this**.

**Example: Nested Class Inheritance:**

class Outer { int outerField = 10; class Inner { int innerField = 20; } } class Subclass extends Outer.Inner { int subclassField = 30; public Subclass(Outer outer) { outer.super(); // Call the constructor of the outer class } }

In this example:

* **Outer** is the outer class.
* **Inner** is a non-static nested class (inner class) within **Outer**.
* **Subclass** extends **Outer.Inner**, inheriting the **innerField** from **Inner**.
* The constructor of **Subclass** explicitly calls the constructor of **Outer** using **outer.super()**, which is possible because **Inner** has an implicit reference to the **Outer** object.

**3. Accessing Enclosing Class Members:**

* A nested class can access the members (fields and methods) of its enclosing class, including private members, as if they were its own.
* The syntax to access an enclosing class member from a nested class is **OuterClassName.this.member**.

**Example: Accessing Enclosing Class Members:**

class Outer { private int outerField = 10; class Inner { void accessOuterField() { int value = Outer.this.outerField; // Access outerField from Outer class System.out.println("Accessed outerField: " + value); } } }

In this example, the **Inner** class accesses the private **outerField** of the **Outer** class using **Outer.this.outerField**.

**4. Static Nested Classes:**

* Static nested classes are not inherently tied to an instance of the enclosing class and do not inherit a reference to an outer class object.
* They can be instantiated independently and do not have access to non-static members of the outer class.
* Inheritance of static nested classes follows the usual class inheritance rules.

**Example: Static Nested Class Inheritance:**

class Outer { static class Nested { static int nestedField = 100; } } class Subclass extends Outer.Nested { int subclassField = 200; }

In this example, **Subclass** extends **Outer.Nested**, inheriting the **nestedField** from **Nested**. Since **Nested** is a static nested class, it doesn't have a reference to an outer class object.

In summary, nested classes in Java can participate in inheritance like any other classes. They can extend other classes, inherit members, and access enclosing class members when necessary. Understanding the relationship between nested classes and inheritance is important for designing and implementing complex class structures in Java.

**KT0213 Final Classes and Inheritance**

In Java, a **final** class is a class that cannot be extended or subclassed. When a class is declared as **final**, it means that no other class can inherit from it. This has several implications for inheritance:

1. **Cannot Be Extended:** A **final** class cannot serve as a parent class for other classes. You cannot create subclasses or child classes of a **final** class. Attempting to do so will result in a compilation error.

final class FinalClass { // Class definition } // This will result in a compilation error class Subclass extends FinalClass { // Class definition }

1. **Prevents Method Overriding:** In a **final** class, methods defined within that class cannot be overridden by subclasses. All methods declared in a **final** class are implicitly **final** as well, which means they cannot be further overridden.

final class FinalClass { void someMethod() { // Method definition } } class Subclass extends FinalClass { // This will result in a compilation error since someMethod() is implicitly final. }

1. **Data Immutability:** **final** classes are often used to create immutable classes, where the state of the object cannot be changed after it is constructed. By making the class itself **final** and its fields **final**, you ensure that the object remains in a consistent state throughout its lifetime.

final class ImmutableClass { private final int value; public ImmutableClass(int value) { this.value = value; } public int getValue() { return value; } }

1. **Performance Benefits:** **final** classes and methods can sometimes offer performance benefits because the compiler and runtime can optimize code more aggressively when they know that classes and methods are not subject to modification or subclassing.

In summary, a **final** class in Java is one that cannot be extended or subclassed. This restriction is useful for creating classes that should not be altered or extended by other classes. It ensures that the class and its methods remain consistent and provides some performance benefits. However, it's important to carefully consider when to use **final** classes, as they limit the flexibility of your class hierarchy.

**KT0214 Abstract Classes and Inheritance**

Abstract classes and inheritance are closely related concepts in object-oriented programming (OOP), particularly in Java. Abstract classes serve as a foundation for inheritance hierarchies and provide a blueprint for derived classes. Let's explore how abstract classes and inheritance work together:

**1. Abstract Classes:**

* An abstract class in Java is a class that cannot be instantiated (i.e., you cannot create objects of an abstract class).
* Abstract classes are designed to be inherited by other classes, and they can have both abstract (unimplemented) methods and concrete (implemented) methods.
* Abstract methods are declared without a body and are meant to be implemented by any concrete (non-abstract) subclass.
* Abstract classes are defined using the **abstract** keyword.

**2. Inheritance with Abstract Classes:**

* Abstract classes are often used as base or parent classes in inheritance hierarchies.
* Subclasses (derived classes) of an abstract class must provide implementations for all abstract methods defined in the abstract class.
* Concrete methods in the abstract class can be inherited by subclasses as-is or optionally overridden.

**Example: Abstract Class and Inheritance:**

abstract class Shape { // Abstract method (no implementation) abstract double area(); // Concrete method void display() { System.out.println("This is a shape."); } } class Circle extends Shape { private double radius; public Circle(double radius) { this.radius = radius; } // Implementing the abstract method double area() { return Math.PI \* radius \* radius; } }

In this example, **Shape** is an abstract class with an abstract method **area()** and a concrete method **display()**. The **Circle** class inherits from **Shape** and provides an implementation for the **area()** method.

**3. Polymorphism with Abstract Classes:**

* Abstract classes and inheritance allow for polymorphism, where objects of different subclasses can be treated uniformly through the use of a common abstract base class.
* You can create references of the abstract class type and assign objects of concrete subclasses to those references, allowing you to call methods defined in the abstract class on objects of different subclasses.

**Example: Polymorphism with Abstract Classes:**

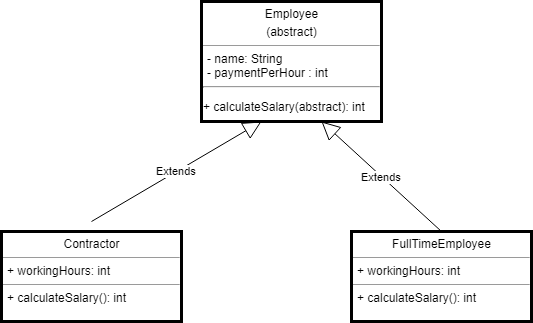
Shape shape1 = new Circle(5.0); // Polymorphism Shape shape2 = new Rectangle(4.0, 6.0); // Polymorphism System.out.println("Area of shape1: " + shape1.area()); System.out.println("Area of shape2: " + shape2.area());

In this example, **shape1** and **shape2** are references of type **Shape** that point to objects of different subclasses (**Circle** and **Rectangle**). The **area()** method is called on each reference, and the appropriate implementation is executed based on the actual object type.

Abstract classes and inheritance provide a powerful mechanism for creating class hierarchies, sharing common behavior, and achieving polymorphism in Java programs. They allow you to design flexible and extensible object-oriented systems.

**KT0215 Java Abstract Class**

* **Creating Recipes with Template M**



In Java, an abstract class is a class that cannot be instantiated directly and is often used as a base class for other classes. Abstract classes can define abstract methods, which are methods without implementations, and concrete methods with implementations. They serve as a blueprint for creating subclasses that provide concrete implementations for the abstract methods. Let's create an example of an abstract class for creating recipes with a template method pattern:

// Abstract class for creating recipes public abstract class Recipe { // Template method that defines the recipe public final void cook() { prepareIngredients(); cookIngredients(); serve(); } // Abstract methods to be implemented by subclasses protected abstract void prepareIngredients(); protected abstract void cookIngredients(); // Concrete method with a default implementation protected void serve() { System.out.println("Serve the dish."); } } // Concrete subclass for a specific recipe public class SpaghettiRecipe extends Recipe { @Override protected void prepareIngredients() { System.out.println("Prepare spaghetti, tomato sauce, and meatballs."); } @Override protected void cookIngredients() { System.out.println("Boil the spaghetti, simmer the sauce, and cook the meatballs."); } // You can override the serve method if needed @Override protected void serve() { System.out.println("Serve delicious spaghetti."); } } // Concrete subclass for another recipe public class PancakeRecipe extends Recipe { @Override protected void prepareIngredients() { System.out.println("Prepare flour, eggs, milk, and sugar."); } @Override protected void cookIngredients() { System.out.println("Mix the ingredients and cook pancakes on a griddle."); } } public class Main { public static void main(String[] args) { Recipe spaghetti = new SpaghettiRecipe(); Recipe pancakes = new PancakeRecipe(); System.out.println("Making Spaghetti:"); spaghetti.cook(); System.out.println("\nMaking Pancakes:"); pancakes.cook(); } }

In this example, we have an abstract class **Recipe** that defines a template method **cook()**. The **cook()** method follows a fixed sequence of steps (prepare ingredients, cook ingredients, serve) that every recipe should follow. The **prepareIngredients()** and **cookIngredients()** methods are declared as abstract, meaning that concrete subclasses must provide their implementations.

We also have two concrete subclasses, **SpaghettiRecipe** and **PancakeRecipe**, each of which implements the abstract methods with recipe-specific instructions.

When we create instances of these subclasses and call the **cook()** method, it follows the template defined in the abstract class but executes the concrete steps provided by the subclasses. This demonstrates the template method pattern, where the overall structure of the algorithm is defined in the abstract class, but the specific steps are implemented in concrete subclasses.

**KT0216 Java Interface**

* **How to think about interface**
* **Complex Algorithm - API defined by extern**
* **Unimplemented methods**

In Java, an interface is a fundamental construct that defines a contract of behavior that implementing classes must adhere to. Interfaces are often used to define a set of methods that a class must implement. Here's how to think about interfaces, their purpose, and their characteristics:

**1. How to Think About Interfaces:**

* **Contract:** An interface is like a contract that specifies a set of methods that implementing classes must provide. If a class implements an interface, it agrees to adhere to the contract by implementing all the methods defined in the interface.
* **Blueprint:** Think of an interface as a blueprint or template for a class. It defines the method signatures that any class implementing the interface must follow.
* **Multiple Inheritance:** Unlike classes, which can have only one superclass, a class can implement multiple interfaces. This enables a form of multiple inheritance in Java, where a class can inherit behavior from multiple sources.

**2. Complex Algorithm - API Defined by Extern:**

* Interfaces are often used to define APIs (Application Programming Interfaces) for complex algorithms or services provided by external libraries or frameworks.
* For example, you might have an interface **SortingAlgorithm** that defines methods like **sort()** and **search()** that any sorting algorithm class must implement.
* Implementing classes can provide their own implementations of the sorting algorithm while adhering to the contract defined by the interface.

**Example: Interface for Sorting Algorithms:**

// Interface defining a sorting algorithm contract public interface SortingAlgorithm { void sort(int[] arr); int search(int[] arr, int target); } // A class implementing the SortingAlgorithm interface public class QuickSort implements SortingAlgorithm { @Override public void sort(int[] arr) { // Implement quicksort algorithm } @Override public int search(int[] arr, int target) { // Implement binary search algorithm } }

In this example, the **SortingAlgorithm** interface defines a contract for sorting and searching methods. The **QuickSort** class implements this interface and provides its own implementations of the sorting and searching algorithms.

**3. Unimplemented Methods:**

* One key characteristic of interfaces is that they can declare methods without providing implementations. These methods are implicitly abstract and must be implemented by any class that implements the interface.
* Implementing classes are required to provide concrete implementations for all methods declared in the interface.

**Example: Interface with Unimplemented Method:**

// Interface with an unimplemented method public interface Logger { void log(String message); } // A class implementing the Logger interface public class ConsoleLogger implements Logger { @Override public void log(String message) { System.out.println("Logging to console: " + message); } }

In this example, the **Logger** interface defines a method **log(String message)** without providing an implementation. The **ConsoleLogger** class implements this interface and provides its own implementation of the **log** method.

Interfaces are a powerful tool for defining contracts, promoting code reusability, and achieving abstraction in Java. They enable you to specify the behavior that classes must adhere to while allowing different classes to provide their own implementations.

**KT0217 Java Interface vs Abstract Class - A Comparison**

In Java, both interfaces and abstract classes are used to define abstractions and establish contracts for implementing classes. However, they have distinct characteristics and use cases. Here's a comparison between Java interfaces and abstract classes:

**1. Definition:**

* **Interface:** An interface is a purely abstract construct in Java. It defines a contract of methods that implementing classes must provide. Interfaces can only declare method signatures, and all methods are implicitly abstract.
* **Abstract Class:** An abstract class is a class that can have both abstract (unimplemented) methods and concrete (implemented) methods. Abstract classes can also have instance variables.

**2. Multiple Inheritance:**

* **Interface:** A class can implement multiple interfaces, allowing it to inherit behavior from multiple sources. Java supports multiple interface inheritance.
* **Abstract Class:** A class can have only one superclass (single inheritance). If a class extends an abstract class, it cannot extend another class, abstract or not.

**3. Constructor:**

* **Interface:** Interfaces cannot have constructors because they cannot be instantiated. Implementing classes provide their own constructors.
* **Abstract Class:** Abstract classes can have constructors. These constructors are called when an instance of a concrete subclass is created.

**4. Method Implementation:**

* **Interface:** All methods in an interface are implicitly abstract and must be implemented by any class that implements the interface. There are no default implementations.
* **Abstract Class:** Abstract classes can have both abstract and concrete methods. Concrete methods provide default implementations that can be inherited by subclasses. Subclasses can choose to override or use these implementations.

**5. Access Modifiers:**

* **Interface:** Interface methods are implicitly **public** and **abstract**. Variables are implicitly **public**, **static**, and **final** (constants).
* **Abstract Class:** Abstract class methods and variables can have various access modifiers, including **public**, **protected**, **private**, or package-private.

**6. Purpose and Use Cases:**

* **Interface:** Interfaces are used to define contracts that classes must adhere to. They are often used to achieve multiple inheritance, create APIs, and define common behavior for unrelated classes.
* **Abstract Class:** Abstract classes are used to provide a common base for related classes. They can contain shared implementation details and provide a partial implementation of a class hierarchy.

**7. Example:**

* **Interface Example:**

interface Drawable { void draw(); } class Circle implements Drawable { public void draw() { // Implementation for drawing a circle } }

* **Abstract Class Example:**

abstract class Shape { abstract void draw(); // Abstract method void move() { // Default implementation for moving a shape } } class Circle extends Shape { void draw() { // Implementation for drawing a circle } }

In summary, interfaces and abstract classes serve different purposes and have different characteristics in Java. Use interfaces when you want to define a contract for unrelated classes or achieve multiple inheritance. Use abstract classes when you want to provide a common base for related classes, including shared implementation details. The choice between them depends on the specific requirements of your design.

**KT0218 Polymorphism**

Polymorphism is one of the fundamental concepts in object-oriented programming (OOP), and it plays a crucial role in Java. It allows objects of different classes to be treated as objects of a common superclass or interface, providing a unified way to work with objects regardless of their specific types. There are two main types of polymorphism in Java: compile-time (or static) polymorphism and runtime (or dynamic) polymorphism.

**1. Compile-Time (Static) Polymorphism:**

* Compile-time polymorphism occurs when the method or operator to be invoked is determined at compile time based on the method's or operator's name and the arguments passed to it.
* It is also known as method overloading or operator overloading.
* Method overloading happens when multiple methods in the same class have the same name but different parameter lists (number or types of parameters).
* The appropriate method to call is determined by the compiler at compile time based on the method signature.
* Example of method overloading:

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

**2. Runtime (Dynamic) Polymorphism:**

* Runtime polymorphism occurs when the method to be invoked is determined at runtime, based on the actual type of the object on which the method is called.
* It is also known as method overriding.
* Method overriding happens when a subclass provides a specific implementation for a method that is already defined in its superclass.
* The appropriate method to call is determined at runtime, based on the actual object type (polymorphic behavior).
* Example of method overriding:

class Animal { void makeSound() { System.out.println("Animal makes a sound"); } } class Dog extends Animal { @Override void makeSound() { System.out.println("Dog barks"); } }

**3. Polymorphic Behavior:**

* Polymorphism allows you to write code that works with objects of various types in a uniform way.
* You can create references of a superclass or interface type and assign objects of different subclasses or implementing classes to those references.
* The method or behavior executed at runtime depends on the actual object type, not the reference type.

**Example of Polymorphism:**

Animal myAnimal = new Dog(); // Polymorphic assignment myAnimal.makeSound(); // Calls Dog's makeSound() method

In this example, **myAnimal** is a reference of type **Animal** that is assigned an instance of **Dog**. When **makeSound()** is called, it invokes the **makeSound()** method of the **Dog** class, demonstrating runtime polymorphism.

Polymorphism simplifies code, promotes code reusability, and allows for the creation of flexible and extensible software systems. It's a core concept in OOP and a key feature of Java.

**KT0219 Introduction to abstraction**

**Abstraction** is one of the fundamental principles of object-oriented programming (OOP). It is the process of simplifying complex systems by breaking them into smaller, more manageable parts, while hiding the unnecessary details from the user. Abstraction allows you to focus on what an object does rather than how it does it. Here are the key aspects of abstraction:

**1. Abstraction in OOP:**

* In OOP, abstraction is implemented using classes and objects.
* A class is an abstract blueprint for creating objects, while an object is an instance of a class.
* Classes define the structure and behavior of objects by specifying attributes (fields) and methods (functions).
* Abstraction allows you to create classes and objects that represent real-world entities, modeling their essential properties and behaviors while hiding the implementation details.

**2. Encapsulation and Information Hiding:**

* Abstraction often goes hand in hand with encapsulation, another OOP principle.
* Encapsulation is the practice of bundling data (attributes) and methods (behavior) that operate on the data into a single unit called a class.
* Information hiding is a related concept that emphasizes restricting access to certain details within a class. It allows you to expose only the necessary information to the outside world while keeping the implementation details hidden.

**3. Example of Abstraction:**

* Let's take a simple example of a **Car** class to illustrate abstraction:

public class Car { // Attributes (data) private String make; private String model; private int year; // Constructor public Car(String make, String model, int year) { this.make = make; this.model = model; this.year = year; } // Method to start the car public void start() { System.out.println("The car is starting."); } // Method to stop the car public void stop() { System.out.println("The car is stopping."); } }

* In this example, the **Car** class abstracts the concept of a car. It defines attributes (make, model, year) and methods (start, stop) to represent the essential properties and behaviors of a car.
* The implementation details of how the car starts or stops are hidden from the user of the **Car** class, promoting abstraction.

**4. Benefits of Abstraction:**

* Abstraction simplifies complex systems by breaking them into manageable components.
* It promotes code reusability because classes and objects can be reused in different parts of an application or in different applications.
* It enhances maintainability by encapsulating implementation details. Changes to the implementation do not affect the code that uses the abstraction.
* Abstraction helps in designing software that is more focused on high-level concepts and less concerned with low-level details.

Abstraction is a powerful concept in OOP that allows you to create models of real-world entities and systems, making your code more organized, maintainable, and understandable. It enables you to work at a higher level of abstraction, emphasizing what objects do rather than how they do it.

**KT0220** **Introduction to Java constructors**

* **Rules for creating a constructor**
* **Constructor overload in Java**

**Introduction to Java Constructors:**

In Java, a constructor is a special type of method that is used to initialize objects when they are created. Constructors have the same name as the class in which they are defined, and they do not have a return type, not even **void**. Constructors are invoked when you create an instance (object) of a class using the **new** keyword.

**Rules for Creating a Constructor in Java:**

* The constructor name must be the same as the class name.
* Constructors do not have a return type, not even **void**.
* You can have multiple constructors in a class with different parameter lists (constructor overloading).
* If you do not provide any constructors in your class, Java provides a default constructor with no arguments (parameterless constructor). However, if you define any constructor yourself, the default constructor is not automatically provided.

**Constructor Overloading in Java:**

Constructor overloading is the practice of defining multiple constructors in a class, each with a different parameter list. This allows objects of the class to be initialized in different ways. You can have constructors with various combinations of parameters, providing flexibility for object creation.

**Example of Constructor Overloading:**

public class Person { private String name; private int age; // Constructor with no parameters public Person() { this.name = "Unknown"; this.age = 0; } // Constructor with name parameter public Person(String name) { this.name = name; this.age = 0; // Default age } // Constructor with name and age parameters public Person(String name, int age) { this.name = name; this.age = age; } // Getter methods public String getName() { return name; } public int getAge() { return age; } }

In this example, the **Person** class has multiple constructors:

* The first constructor initializes the name to "Unknown" and age to 0.
* The second constructor allows you to specify the name while setting the age to the default value of 0.
* The third constructor accepts both name and age parameters and initializes the object accordingly.

Constructor overloading provides flexibility when creating **Person** objects. You can create a **Person** object with just a name, with a name and age, or with no parameters, and the appropriate constructor will be called based on the arguments provided.

Constructor overloading is a useful technique in Java to create objects with different initial states and to provide flexibility for object creation in different scenarios.

**KT0221 Constructor chaining**

**Constructor chaining** in Java refers to the process of one constructor calling another constructor in the same class or in its superclass. This allows you to reuse code and avoid duplicating initialization logic across multiple constructors. Constructor chaining is achieved using the **this()** or **super()** keyword, depending on whether you want to call a constructor in the same class or in the superclass.

Here are some key points to understand about constructor chaining:

1. **Using this() to Call Another Constructor in the Same Class:**
   * You can use the **this()** keyword to call another constructor in the same class. This must be the first statement in the constructor.
   * It allows you to reuse initialization logic and avoid code duplication.
2. **Using super() to Call a Constructor in the Superclass:**
   * You can use the **super()** keyword to call a constructor in the superclass. This must also be the first statement in the constructor.
   * It allows you to invoke a constructor in the superclass before executing the code in the subclass's constructor.
   * This is useful when the subclass constructor needs to initialize additional attributes or perform some specific tasks after the superclass's constructor has executed.
3. **Constructor Chaining Rules:**
   * Constructor chaining can involve a sequence of constructors calling each other. However, you should be careful to avoid infinite loops of constructor calls.
   * The chain of constructor calls should ultimately lead to the execution of a constructor in the superclass (either directly or indirectly).
4. **Example of Constructor Chaining:**

class Animal { private String name; // Constructor with name parameter public Animal(String name) { this.name = name; } } class Dog extends Animal { private int age; // Constructor with name and age parameters public Dog(String name, int age) { super(name); // Call Animal's constructor with name this.age = age; } }

In this example, the **Dog** class extends the **Animal** class. The **Dog** class has a constructor that takes both a name and an age parameter. It uses the **super(name)** statement to call the constructor in the **Animal** class, passing the name parameter to it. This ensures that the **name** attribute in the **Animal** class is initialized correctly before the **Dog** class's constructor proceeds to initialize the **age** attribute.

Constructor chaining is a useful technique for organizing and reusing code in class hierarchies. It helps maintain code consistency and reduces redundancy in constructor initialization logic.

**KT0222Understanding basics of Encapsulation with Setter methods**

* **Getters and generating getters and setters with <E>**

**Encapsulation** is one of the fundamental principles of object-oriented programming (OOP). It involves bundling data (attributes) and the methods (getters and setters) that operate on that data into a single unit called a class. The primary goal of encapsulation is to restrict direct access to an object's internal state and provide controlled access through methods. This helps maintain data integrity and ensures that the object's state remains consistent.

In Java, you can achieve encapsulation by following these basic principles:

1. **Declare Class Variables as Private:**
   * Class variables (attributes) should be declared as **private** to hide them from direct access outside the class. This prevents external code from modifying the object's state directly.
2. **Provide Getter and Setter Methods:**
   * Getter methods (also known as accessor methods) are used to retrieve the values of private attributes.
   * Setter methods (also known as mutator methods) are used to modify the values of private attributes.
   * By providing these methods, you control how the internal state of an object is accessed and modified.

Here's an example that illustrates encapsulation with getter and setter methods:

public class Student { // Private attributes private String name; private int age; // Getter method for name public String getName() { return name; } // Setter method for name public void setName(String name) { // You can add validation logic here if needed this.name = name; } // Getter method for age public int getAge() { return age; } // Setter method for age public void setAge(int age) { // You can add validation logic here if needed this.age = age; } }

In this example, the **Student** class encapsulates the **name** and **age** attributes by declaring them as **private**. It provides getter and setter methods for each attribute, allowing controlled access to these attributes.

**Advantages of Encapsulation:**

* Data Hiding: Encapsulation hides the internal state of an object, making it less prone to unintended modifications.
* Controlled Access: You can add validation logic or access control in getter and setter methods.
* Flexibility: You can modify the internal representation of an object without affecting external code that uses it.
* Maintenance: Encapsulation makes it easier to maintain and refactor code.

**Generating Getters and Setters with IDEs:**

Most Integrated Development Environments (IDEs) provide automated tools to generate getters and setters for class attributes. This can save you from writing boilerplate code. Here's how you can generate getters and setters in popular IDEs like Eclipse and IntelliJ IDEA:

**Eclipse:**

* Right-click on the attribute.
* Select "Source" from the context menu.
* Choose "Generate Getters and Setters."
* Select the attributes for which you want to generate getters and setters.
* Click "Generate."

**IntelliJ IDEA:**

* Right-click on the attribute.
* Select "Refactor" from the context menu.
* Choose "Encapsulate Fields."
* Select the attributes for which you want to generate getters and setters.
* Click "Refactor."

These automated tools can significantly speed up the process of encapsulating class attributes with getters and setters.

**KT0223Initialisation of member variables**

In Java, member variables (also known as instance variables) are initialized as part of the object creation process. Member variables are attributes of a class, and each object of the class can have its own values for these variables. There are several ways to initialize member variables:

**1. Default Initialization:**

* When an object of a class is created, Java provides default values for member variables if you don't explicitly initialize them. The default values depend on the data type:
  + Numeric types (int, float, double, etc.): 0
  + Boolean type: false
  + Object references: null
  + Char type: '\u0000' (null character)

**2. Initialization in Constructors:**

* You can initialize member variables in the constructor(s) of the class. Constructors are special methods called when an object is created.
* Use constructor parameters to set the initial values of member variables.

**Example of Initializing Member Variables in Constructors:**

public class Person { private String name; private int age; // Constructor to initialize name and age public Person(String name, int age) { this.name = name; this.age = age; } // Getter methods for name and age public String getName() { return name; } public int getAge() { return age; } }

In this example, the **Person** class has a constructor that takes **name** and **age** as parameters and initializes the corresponding member variables.

**3. Initialization Blocks:**

* Java allows you to create initialization blocks (static and instance) to initialize member variables.
* Instance initialization blocks are executed when an object is created, before constructors.

**Example of Using Instance Initialization Block:**

public class Employee { private String name; private int employeeId; // Instance initialization block { name = "John Doe"; employeeId = 0; } // Constructor public Employee(String name, int employeeId) { this.name = name; this.employeeId = employeeId; } // Getter methods for name and employeeId public String getName() { return name; } public int getEmployeeId() { return employeeId; } }

In this example, an instance initialization block is used to set default values for **name** and **employeeId**.

**4. Initialization Using Setter Methods:**

* You can also provide setter methods to initialize member variables after object creation.

**Example of Using Setter Methods:**

public class Student { private String name; private int age; // Setter methods public void setName(String name) { this.name = name; } public void setAge(int age) { this.age = age; } // Getter methods for name and age public String getName() { return name; } public int getAge() { return age; } }

In this example, you can set the **name** and **age** of a **Student** object using setter methods after the object is created.

The choice of how to initialize member variables depends on your specific requirements and design. Constructors are commonly used for initialization, but you can use other methods as needed for flexibility and code organization.

**KT0224First advantage of encapsulation**

The first advantage of encapsulation in object-oriented programming (OOP) is **data hiding**. Data hiding is a key aspect of encapsulation, and it refers to the practice of restricting direct access to an object's internal state (its attributes or data members) from outside the object. This means that the internal data of an object is hidden and can only be accessed and modified through well-defined methods (getters and setters) provided by the class.

Here are some benefits of data hiding, which is the first advantage of encapsulation:

* **Enhanced Data Security and Integrity:** By making the internal state of an object private and allowing controlled access through methods, you prevent external code from directly modifying the object's data. This enhances data security and ensures that the data remains in a consistent and valid state. Data can only be changed through the methods that encapsulate it, which allows for validation and error checking.
* **Abstraction:** Data hiding allows you to abstract the implementation details of an object. Clients of the class do not need to know how the data is stored or manipulated internally; they only need to interact with the public methods. This simplifies the usage of the class and reduces the complexity of client code.
* **Flexibility in Implementation:** With data hiding, you can change the internal representation of the object's data without affecting the external code that uses the class. This flexibility enables you to optimize the implementation or make improvements without breaking existing code.
* **Code Maintenance:** Encapsulation promotes clean code and makes it easier to maintain. When changes are needed, they can be made in one place within the class's methods, rather than scattered throughout client code.
* **Controlled Access:** Getter and setter methods provide controlled access to the object's data. You can enforce validation rules, access control, and additional logic within these methods. For example, you can ensure that certain attributes are never set to invalid values.

**Example of Data Hiding:**

public class BankAccount { private double balance; // Private member variable // Constructor to initialize the balance public BankAccount(double initialBalance) { // Validation can be added here to ensure initialBalance is non-negative this.balance = initialBalance; } // Getter method to retrieve the balance public double getBalance() { return balance; } // Setter method to deposit funds public void deposit(double amount) { // Validation can be added here to ensure amount is non-negative balance += amount; } // Setter method to withdraw funds public void withdraw(double amount) { // Validation can be added here to ensure amount is non-negative if (amount <= balance) { balance -= amount; } else { System.out.println("Insufficient funds"); } } }

In this example, the **BankAccount** class encapsulates the **balance** attribute and provides getter and setter methods for controlled access. Data hiding ensures that the **balance** is not directly manipulated from outside the class, promoting data integrity and security.

**Internal Assessment Criteria and Weight**

1. IAC0201 Concepts of Object-Oriented Programming in Java are understood.

**References**

***Books:***

1. *"Java: The Complete Reference" by Herbert Schildt:*
   * *This book covers Java programming comprehensively, including OOP concepts and Java's core features.*
2. *"Head First Java" by Kathy Sierra and Bert Bates:*
   * *Known for its engaging style, this book provides a beginner-friendly introduction to Java and OOP principles.*
3. *"Effective Java" by Joshua Bloch:*
   * *A must-read for Java developers, this book offers best practices and design guidelines for writing effective and robust Java code.*
4. *"Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin:*
   * *While not Java-specific, this book focuses on writing clean and maintainable code, which is essential in OOP and Java development.*
5. *"Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides:*
   * *This classic book explores design patterns in OOP, which are reusable solutions to common programming problems.*

***Online Courses:***

1. *Coursera's "Object-Oriented Programming in Java" (by the University of California, San Diego):*
   * *This online course covers the fundamentals of OOP and Java, suitable for beginners.*
2. *edX's "Introduction to Java: Programming and Data Structures" (offered by Microsoft):*
   * *This course provides a comprehensive introduction to Java programming, including OOP concepts and data structures.*
3. *Udemy's "Java Programming Masterclass for Software Developers" (by Tim Buchalka):*
   * *A highly-rated course that covers Java in depth, including OOP principles and advanced topics.*
4. *Codecademy's "Learn Java" Course:*
   * *A beginner-friendly online course that introduces Java and OOP through interactive exercises.*

***Websites and Documentation:***

1. [*Oracle's Java Documentation*](https://docs.oracle.com/en/java/)*:*
   * *The official documentation provides in-depth information about Java's core libraries, language features, and APIs.*
2. [*Java Tutorials on Oracle*](https://docs.oracle.com/javase/tutorial/)*:*
   * *A collection of tutorials covering various aspects of Java, including OOP concepts.*
3. [*Baeldung*](https://www.baeldung.com/)*:*
   * *Baeldung offers tutorials and articles on Java and Spring Framework, which include OOP-related topics.*
4. [*GeeksforGeeks - Java Programming Language*](https://www.geeksforgeeks.org/java-programming-language/)*:*
   * *A vast resource with tutorials and articles on Java programming, including OOP concepts and examples.*
5. [*Stack Overflow*](https://stackoverflow.com/questions/tagged/java)*:*
   * *A community-driven platform where you can find answers to Java-related questions and learn from discussions.*