Course: MSc DS

Java Programming

Module: 2

Learning Objectives:

1. For Java to represent real-world things, you must understand the fundamental ideas behind classes and objects.
2. Understand constructors and their function in the instantiation of objects while examining inheritance to benefit from code reuse.
3. Learn about abstraction and encapsulation to improve security and readability, and get a solid understanding of polymorphism's concepts to encourage flexibility in programming.
4. Understand the differences between abstract classes and interfaces and their importance in permitting multiple inheritance and implementing standardised protocols.

Structure:

2.1 Classes and Objects

Within the domain of object-oriented programming (OOP), Java is often regarded as a prominent and exemplary language. The fundamental principles of Object-Oriented Programming (OOP) and the Java programming language revolve on the interconnected notions of classes and objects. Gaining a comprehensive understanding of these principles is of utmost importance in comprehending Java's methodology for representing and resolving real-world issues via programming.

A Java class might be conceptualised as a blueprint or a template. The process involves establishing the framework of an object by the delineation of properties, often referred to as fields or variables, and behaviours, known as methods or functions. For example, let us assume a hypothetical class referred to as "Car". This course may include the identification and categorisation of characteristics such as hue, manufacturer, and velocity. Furthermore, it may include the characterisation of actions such as acceleration and deceleration. It is vital to comprehend that a class only serves as a framework or a delineation, devoid of possessing any tangible real-world facts in its own right.

Conversely, an object may be seen as the concrete realisation of a class, serving as an embodiment of the abstract blueprint inside the physical realm. Returning to the example of a vehicle, the concept of a 'vehicle' as a class serves to establish the fundamental characteristics and attributes that define what a car is. On the other hand, an instance of the Car class may be seen as a concrete manifestation of a car, such as a Toyota that is coloured red and is moving at a speed of 60 miles per hour. In the Java programming language, the instantiation of objects is achieved by using the 'new' keyword, which is then followed by the invocation of a constructor, a specialised method belonging to the respective class. After an object is instantiated, it acquires its own state, which refers to its attributes, and becomes capable of executing actions, known as methods, that are specified inside its class.

Fundamentally, a class may be seen as a broad and abstract notion, while an object is a particular and concrete manifestation of that notion. The aforementioned differentiation enables Java programmers to establish entities in a more abstract form, namely as classes, and afterwards generate numerous unique instances, referred to as objects, to reflect the various circumstances that may arise in real-world applications. The OOP paradigm of Java is characterised by its remarkable combination of power and simplicity, which greatly enhances its effectiveness in addressing various problem-solving scenarios.

2.2 Constructors

The fundamental basis of Java's object-oriented paradigm is in its capacity to generate instances of classes, therefore transforming abstract conceptual designs into concrete realities. The process of instantiation is supervised by specialised methods referred to as constructors.

Within the Java programming language, a constructor is a distinctive method that has the same name as the class it belongs to and lacks an explicit return type. The automatic invocation of the method occurs upon the instantiation of an object belonging to the class. The fundamental function of a constructor is to initialise the object that has been freshly constructed. The feature allows for the assignment of initial values to an object's attributes or the establishment of certain conditions or states that the object may need immediately after its creation.

Java provides support for many kinds of constructors, hence enabling flexibility in the process of object instantiation. The Java compiler automatically generates a default constructor for a class if no explicit constructor is supplied. The process of initialisation involves assigning default values to all properties, such as zero for integers and null for objects. Frequently, Java developers use parameterised constructors, which receive parameters in order to initialise the object with predetermined values. This feature enables the creation of objects with predetermined beginning states. In the context of a 'Book' class, it is common for a parameterised constructor to accept the title and author as parameters in order to instantiate a distinct book object.

Overloading is an additional aspect of constructors inside the Java programming language. In object-oriented programming, it is possible for a class to possess many constructors, with each constructor varying in terms of the amount or kind of inputs it accepts. The presence of many options for object initialisation allows for flexibility in accommodating diverse situations.

Constructors, despite their apparent simplicity, serve as a crucial component in Java programming. They guarantee that items start their lifespan in a uniform and regulated way. The absence of constructors in object-oriented programming may result in objects being in an indeterminate or unstable state, hence causing unexpected behaviour and potential vulnerabilities inside the system. By including explicit techniques for object initialisation, constructors enhance the resilience and dependability of Java programmes, serving as a fundamental element of secure and organised object-oriented architecture.

2.3 Inheritance

In the realm of object-oriented programming, the concept of inheritance stands as a key principle. It gives Java developers the opportunity to build a connection between two classes, hence facilitating the reuse of code and constructing a hierarchical structure among objects.

The fundamental concept of inheritance involves the transmission of properties and methods from one class to another, establishing a hierarchical connection between a parent class and its child class. The parent class is often referred to as the "superclass" or "base class," while the child class is denoted as the "subclass" or "derived class." The hierarchical structure guarantees the automatic inheritance of attributes and behaviours from the superclass to the subclass. Consequently, software developers have the capability to generate a novel class by building upon an already existing one, so inheriting its attributes and enhancing or replacing them as required.

In the Java programming language, the concept of inheritance is realised via the use of the "extends" keyword. When a class is formed using the "extends" keyword, followed by the name of another class, it acquires the fields and methods of that class via inheritance. As an example, in the context of object-oriented programming, the presence of a base class named "Vehicle" with inherent properties such as speed and weight allows for the possibility of a derived class called "Car" to expand upon "Vehicle" and acquire these attributes without the need to explicitly restate them.

In addition to serving as the basis for property sharing, inheritance in Java enables subclasses to alter or extend the inherited behaviours. The process of doing this is accomplished by means of method overriding, when a subclass furnishes a distinct implementation for a method that has already been specified in its superclass. In Java, the "super" keyword may be used to invoke a method from the superclass that has been overridden.

In addition to mitigating repetition within code, inheritance serves as a structural mechanism for classifying objects according to their common attributes. Java enables developers to enhance the clarity and maintainability of programmes by using a superclass to aggregate common characteristics and methods, and allowing subclasses to inherit and customise them, therefore reflecting real-world connections and hierarchies.

2.4 Polymorphism

The fundamental principle of object-oriented programming centres on the idea of polymorphism, which originates from the Greek terms "poly" (meaning numerous) and "morph" (meaning shapes). Within the context of Java programming, the concept of polymorphism encompasses the notion that a singular interface has the capacity to include a diverse range of types. This enables objects belonging to distinct classes to be seen and manipulated as objects of a common parent class.

Polymorphism is a fundamental concept in Java programming, which is achieved via two main mechanisms: method overloading and method overriding. The use of method overloading enables the inclusion of numerous methods inside a single class that possess the same name, although with distinct arguments. This implies that a single method name has the ability to fulfil many functions depending on the supplied parameters. In contrast, method overriding refers to the process via which a subclass offers a distinct implementation for a method that has already been defined in its superclass. This guarantees that the appropriate version of the function is invoked, taking into account the actual type of the object.

The dynamic nature of polymorphism is a noteworthy aspect. The optimal illustration of this concept occurs when a reference variable of a superclass is used to refer to an object of a subclass. When a method that has been overridden is invoked using the "this" reference, the Java Virtual Machine (JVM) identifies the actual type of the object during runtime and runs the corresponding method. The dynamic method dispatch is a fundamental aspect of Java's runtime polymorphism.

The importance of polymorphism in software design cannot be underestimated. Polymorphism promotes flexibility by enabling the consistent treatment of objects belonging to various classes. Developers have the capability to incorporate new classes into a codebase without making any modifications to the current code, hence facilitating the enhancement of extensibility. In the context of a graphics system, it is possible to implement a general "Shape" class that serves as a base class for more specific derived classes such as "Circle," "Square," and "Triangle." By using the concept of polymorphism, it becomes possible to create methods that may effectively manipulate a Shape reference, so effortlessly accommodating objects originating from any class.

2.5 Abstraction and Encapsulation

Within the domain of software design, particularly in the context of Java programming, abstraction may be likened to the process of extracting the core essence of an object, emphasising its key functionality, while simultaneously disregarding the finer intricacies. The aforementioned methodology facilitates the comprehension of complex systems by their division into comprehensible components, hence enabling developers to build upon levels of abstraction. Consider an automobile. The driver is not required to possess extensive knowledge of combustion processes or the inner workings of the gearbox. Their proficiency is limited to operating the vehicle, which may be considered a kind of high-level abstraction.

In the Java programming language, the concept of abstraction is implemented via the use of interfaces and abstract classes. An abstract class has the capability to have abstract methods, which are methods without implementation details. This establishes a standardised framework for subsequent classes to derive from, guaranteeing the implementation of certain methods. In contrast, interfaces increase the level of abstraction. Method signatures are established without any accompanying implementation, ensuring that derived classes will be responsible for furnishing the specific functionality of these methods.

The concept of abstraction involves the process of simplification, whereas encapsulation pertains to the act of shielding. Object encapsulation refers to the process of combining data properties and procedures into a cohesive unit or class, while also imposing limitations on the accessibility of certain components of the object. The presence of this protective layer guarantees the concealment of sensitive data from external tampering and inadvertent modifications, whilst revealing only essential information via publicly accessible channels.

The encapsulation mechanism in Java is implemented via the use of access modifiers. When an attribute is designated as 'private', it is not possible to directly access or modify it from external sources outside of the class. In contrast, the use of 'public' getter and setter methods enables regulated access to those attributes, hence guaranteeing the ability to verify conditions or execute modifications prior to the retrieval or alteration of data.

2.6 Interfaces and Abstract Classes

The usage of interfaces and abstract classes in Java serves to emphasise the Object-Oriented Programming (OOP) concepts. Both serve as fundamental frameworks from which specific classes may be generated, but, their application and purpose differ significantly.

Abstract classes serve as an intermediary between a fully implemented class and a purely abstract design. In object-oriented programming, it is possible for classes to include both abstract methods, which lack implementation details, and concrete methods, which have a defined body of code. This dual nature enables developers to include predefined functionality while allowing for the customisation of some aspects via the use of derived classes. since an example, an abstract class called 'Vehicle' may include an abstract method called 'startEngine()' that does not provide specific details on how each vehicle starts its engine, since this process may differ across different types of vehicles like as cars, boats, or aeroplanes.

In contrast, interfaces embody a purer manifestation of abstraction. The interfaces just consist of method signatures, lacking any implementation details, hence necessitating the implementing classes to provide the exact implementations. The interface 'Flyable' is defined with a function 'fly()', which requires any class that implements this interface, such as 'Bird' or 'Aeroplane', to specify the specific mechanisms and processes involved in their respective modes of flight. In addition, Java provides the capability for a class to implement numerous interfaces, hence facilitating a versatile implementation of multiple inheritance, a feature that is not present in abstract or regular classes.

The selection between interfaces and abstract classes is contingent upon the specific requirements of the design. Abstract classes are often used when there is a need to provide a shared foundation that includes pre-implemented methods. However, if the objective is to precisely establish a contract that can be followed by several classes without the associated complexities of an inheritance tree, interfaces are the superior option.

2.7 Summary

❖ Module 2 explores the fundamental ideas behind Java's Object-Oriented Programming paradigm. The fundamental nature of Java revolves on the concepts of classes and objects, whereby classes serve as abstract representations and objects represent concrete examples of these abstractions. The objects are animated via the use of constructors, which establish their starting state and guarantee their compliance with predetermined criteria.

- Expanding upon this fundamental concept, inheritance assumes a crucial function by facilitating the derivation of attributes and behaviours of new classes from pre-existing ones, hence fostering the reuse of code and the establishment of a hierarchical arrangement. Polymorphism enhances this capability by enabling objects from disparate classes to be regarded as though they are members of the same class, hence promoting flexibility in programming. The concepts of abstraction and encapsulation are closely related. Abstraction involves concealing intricate details and providing a simplified interface, while encapsulation involves combining data and processes into a cohesive entity to protect information.
- In conclusion, interfaces and abstract classes provide a systematic way for defining contracts and partial implementations, respectively. Design patterns play a crucial role in governing the interactions among various components of a software programme, establishing guidelines that must be followed by certain classes. Collectively, these components provide the fundamental

principles of Java's Object-Oriented Programming (OOP), hence facilitating the development of resilient, manageable, and optimised coding methodologies.

2.8 Keywords

- **Classes:** The blueprints pertain to the construction of things, including both dataand procedures inside them.
- **Objects:** The programme contains instances of classes that represent thingsfound in the actual world.
- **Constructors:** Special methods in a class responsible for initialising object state.
- **Inheritance:** The mechanism facilitating the inheritance of attributes and behaviours from one class to another.
- **Polymorphism:** The concept of object polymorphism refers to the capability of objects belonging to distinct kinds to be manipulated and used as if they were objects of a common type.
- **Encapsulation:** Binding together of data and functions, protecting them from external interference.

2.9 Self-Assessment Questions

1. Describe what distinguishes a class from an object. Give an illustration to back upyour statement.
2. Describe how constructors are used in object-oriented programming and theirsignificance. A class may have more

than one constructor. Elaborate.

3. How does inheritance improve Java's ability to reuse code? Give a scenario wheninheriting would be advantageous.
4. Explain polymorphism by describing a situation in the real world whenintroducing polymorphism would be beneficial.
5. Compare and contrast the ideas of abstraction and encapsulation. Why are theyregarded as the fundamental tenets of object-oriented design?

2.10 Case Study

Title: Designing a Library Management System with OOP PrinciplesIntroduction:

Given the rapid increase in the quantity of books and digital materials, contemporary libraries need a resilient system to effectively oversee and organise their resources. The inadequacy of traditional approaches has prompted the need for a shift towards computerised solutions. Explore the domain of Object-Oriented Programming in Java, presenting a sophisticated resolution to this particular challenge.

Case Study:

The Townsville Public Library, a library of moderate size, has challenges due to its antiquated card catalogue system. There has been a discernible rise in instances of errors, delays, and inefficiencies that have been observed. Given the library's intention to enhance its collection and undertake digitisation efforts, there arises a need for the implementation of a more sophisticated system.

Background:

The library has a collection of more than 50,000 volumes, as well as a wide range of digital resources and a regular schedule of activities. The institution caters to a large number of individuals, each exhibiting diverse patterns of borrowing behaviour. The previous method used a manual approach, using physical records and cards for every book, resulting in a protracted procedure for locating, borrowing, and returning books.

Your Task:

You've been hired as a Java developer to design a new library management system. In order to design a system that is fast and scalable, it is imperative to use object-oriented programming (OOP) concepts such as inheritance, polymorphism, encapsulation, and abstraction.

Questions to Consider:

1. How can the classes and objects be organised to effectively represent books, digital resources, members, and personnel inside the system?
2. Which object-oriented programming (OOP) concepts are best applicable in the design of the borrowing and returning functions?
3. How would inheritance play a role in categorising

different types of books or resources?

4. Considering scalability, how would interfaces and abstract classes assist in future expansions or modifications of the system?

Recommendations:

In order to establish an effective library management system, the implementation of a well-defined hierarchical class structure is of utmost importance. The fundamental classes, such as 'Book', 'Member', and 'Staff', possess the capability to be further subclassed in order to accommodate additional particular categories. The concept of encapsulation serves to maintain the integrity of data, whereas abstraction functions to conceal intricate activities. Interfaces may serve the purpose of establishing borrowing restrictions or loan lengths, hence offering adaptability for prospective modifications.

Conclusion:

The object-oriented programming (OOP) concepts in Java provide a systematic methodology for the construction of complex systems. By comprehending and proficiently implementing these ideas, developers have the capability to construct scalable, efficient, and flexible systems like the newly developed management system of the Townsville Public Library.

2.11 References

1. Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C. and Mergen, M., 1999. Implementing jalapeño in java. ACM SIGPLAN Notices, 34(10), pp.314-324.
2. Deitel, P.J., 2002. Java how to program. Pearson Education India.
3. Schildt, H., 2003. Java™ 2: A Beginner's Guide.
4. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G. and Ur, S., 2002. Multithreaded Java program test generation. IBM systems journal, 41(1), pp.111-122.
5. Eckel, B., 2003. Thinking in JAVA. Prentice Hall Professional.