AWS Lambda is a cloud-based service that allows you to run code without provisioning or managing servers. Lambda executes your code only when needed and scales automatically to run your code with high availability. It enables developers to build applications and services quickly without having to manage servers. You can use Lambda to build serverless applications, including backend services.

**Use of lambda:** for example, an image is uploaded in s3 bucket in a wrong format and AWS lambda is triggered on time an object is added to the bucket and image is processed and converted into the templates based on the device reading the data.

How lambda works: When a client data to lambda (could be anyone sending request to lambda could be aws services). Depending on the size of the data or volume of data it runs on the defined number of containers. If it is less requests than can run on single container and if large can run on multiple containers. A container contains the code that the user has provided to satisfy the query.

When an event is triggered, Lambda will execute your code. It spins up the necessary computing resources, runs your code, and then tears down the resources after it’s done. You only pay for the computing time you consume- there is no charge when your code is not running.

The following are some of the features of AWS Lambda:

Serverless: AWS Lambda is a serverless compute service that runs your code in response to events and automatically manages the underlying compute resources for you.

Scaling: AWS Lambda can scale to hundreds of thousands of concurrent executions, allowing you to build highly parallel, event-driven applications.

* Event-Driven: AWS Lambda enables you to respond to events such as file uploads and changes to data in an Amazon S3 bucket.
* Cost-Effective: With AWS Lambda, you only pay for the computing time you use. There is no charge when your code is not running.
* Flexibility: AWS Lambda supports a variety of languages and runtimes, including Node.js, Java, Python, and C#. This allows you to use the language and libraries of your choice.

Polymorphism

### Advantages of Polymorphism in Java

1. Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
2. Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.
3. Supports dynamic binding, enabling the correct method to be called at runtime, based on the actual class of the object.
4. Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

### Disadvantages of Polymorphism in Java

1. Can make it more difficult to understand the behavior of an object, especially if the code is complex.
2. This may lead to performance issues, as polymorphic behavior may require additional computations at runtime.

In Java, **encapsulation** is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent

1. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.
2. **Multilevel Inheritance:**In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

### **Multiple Inheritance: When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.**

### **Hierarchical Inheritance:** **When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.**

### **Hybrid Inheritance:** **Inheritance consisting of multiple types of inheritance is called hybrid inheritance.**

Dataframe: In [**Spark**](https://intellipaat.com/blog/what-is-apache-spark/), DataFrames are the distributed collections of data, organized into rows and columns. Each column in a DataFrame has a name and an associated type. DataFrames are similar to traditional database tables, which are structured and concise. We can say that DataFrames are relational databases with better optimization techniques.

Spark DataFrames can be created from various sources, such as Hive tables, log tables, external databases, or the existing [**RDDs**](https://intellipaat.com/blog/tutorial/spark-tutorial/programming-with-rdds/). DataFrames allow the processing of huge amounts of data.

### Features of DataFrames: Some of the unique features of DataFrames are:

* **Use of Input Optimization Engine**: DataFrames make use of the input optimization engines, e.g., **Catalyst Optimizer**, to process data efficiently. We can use the same engine for all Python, Java, Scala, and R DataFrame APIs.
* **Handling of Structured Data**: DataFrames provide a schematic view of data. Here, the data has some meaning to it when it is being stored.
* **Custom Memory Management**: In RDDs, the data is stored in memory, whereas DataFrames store data off-heap (outside the main Java Heap space, but still inside RAM), which in turn reduces the garbage collection overload.
* **Flexibility**: DataFrames, like RDDs, can support various formats of data, such as CSV, [Cassandra](https://intellipaat.com/blog/apache-cassandra-a-brief-intro/), etc.
* **Scalability**: DataFrames can be integrated with various other Big Data tools, and they allow processing megabytes to petabytes of data at once.

**Boards**: Boards are the key element of using Contour, allowing the user to perform analysis and filters on the input dataset or join with another dataset. To manipulate the data, we will apply a series of boards to it. Each board provides a different possible visualization and/or data manipulation action.

Boards have a few different capabilities:

* **Visualize**: Display your data in different forms (a histogram, a heatmap with geographic data).
* **Filter rows**: Filter your dataset using regular expressions or with built-in date, number, and null comparisons.
* **Aggregate**: Calculate aggregate metrics about your data (for example, calculate the mean value of data in a column).
* **Manipulate columns**: Add columns from other datasets, derive new columns or remove unneeded columns.
* **Combine with other datasets**: Join to or perform set math operations with other datasets.

**Histogram:** Many boards which visualize data also allow for interactive filtering. For example, in a Histogram board, clicking one or more bars will filter the active dataset. When a filter is applied within a board in such a way, the filter summary at the bottom of the board will describe the active filter

**Pivot:** Some boards that allow you to calculate aggregate metrics have an option to pivot. This switches your working dataset to the aggregate data computed in that board, instead of the original dataset. The new dataset will include the columns selected in the aggregate board, as well as one column for every aggregate.

**Joining Datasets:** Contour provides a unique ability to allow users to join data from different sources across the ecosystem together.

**Expression:** board lets you work with Contour’s rich expression language to derive new columns from your data, perform complex filtering or perform various aggregations.

**Data Lineage:** is an interactive tool that provides a holistic view of data pipelines within the Foundry platform. With so many ways to change datasets within Foundry, you may think it would be difficult to monitor and administer pipelines. However, Foundry includes a component called Data Lineage which allows you to examine a dataset's lineage and easily view the transformations done throughout the pipeline that produced it.

**Reports** is an application that creates documents and dashboards to share your analytical work. Reports brings all of your Foundry analysis work together in one place, integrating seamlessly with platform analysis tools like Contour. Use Reports to write collaborative documents; share charts and observations about your data; or create long-lived, **real-time** dashboards of important metrics.

**Code Repositories:** Code Repositories is an application in the Foundry platform that enables you to write code-based transformations of data. Its front-end provides a simple integrated development environment that supports a variety of programming languages such as SQL, Python, and Java.

**Benefits:** Code Repositories provides code writers with the benefits of [git](https://git-scm.com/), particularly history and branches, as they author their data pipeline code. Foundry also applies these same concepts to the data that is then generated by that code. Foundry datasets have internal branches, and through git permits you to modify your data pipeline code in branches and run the data through those transformations using the exact same system as your production pipeline.

**An ontology** is a categorization of the world. It includes objects (which can be entities or events), their properties, and the relations between these entities. To create an ontology in Foundry, objects and the relationships between objects must be defined, which is accomplished by definitions called “Object Types” (the definitions of the entities), “Properties” (features or characteristics of Object Types), and “Relations” (the definitions of what relationships exist between different Object Types).

Object types are comprised of **properties** and **metadata**.

A **property** is an attribute that defines an object. For example, the object type Employee might have properties like name, employee\_id, start\_date, and role. When considered as a whole, these make up the characteristics of an employee.

The **metadata** used for object types desc