DROOLS 8

MODULE 01:
DROOLS OVERVIEW

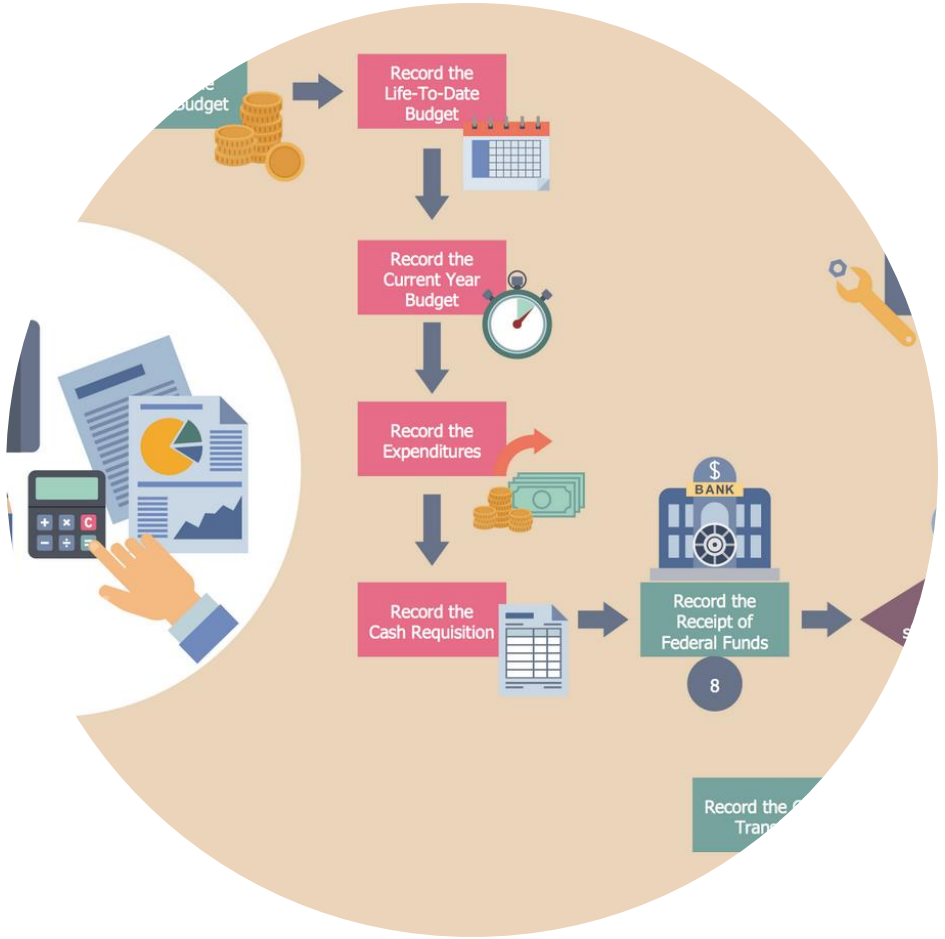
INTRODUCTION TO RULES ENGINES

Presented by John Paul Franke

Course Overview

Welcome to Module 01 of our exploration into the world of Drools 8.4, a powerful business rule management system. In this module, we will be covering the fundamentals of rule-based applications and explore how Drools enables efficient and flexible business rules implementation.

We'll be discussing the components that make up Drools, the benefits of using a rule engine, and some key features that have been introduced in version 8.4. By the end of this module, you'll gain a solid foundation on which to build your skills in complex rule management and automation, enabling you to deliver solutions that meet evolving business requirements with agility and precision.



Learning Objectives

In this module, we will aim to understand the core concepts of Drools 8.4.

- Get to know what a Rule Engine does and why it's useful.
- Look back at how Rule Engines have grown and changed over time.
- Dive into what makes Drools a go-to choice for managing business rules.
- See how AI and Rule Engines work together to make smart decisions.
- Set up your own space for working with Drools.
- Check out examples of how Drools is used in the real world.
- Engage in hands-on activities to bring theory into practice.
- Get ready for quizzes that will test what you've learned.

Defining a Rule Engine

Simplifying Complex Decisions

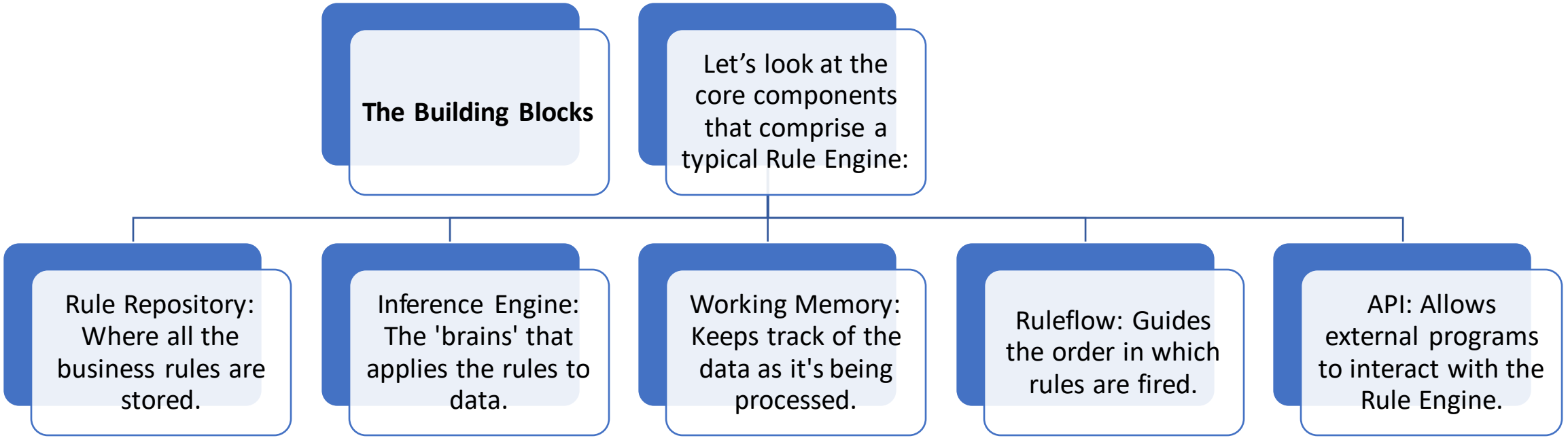
A rule engine is a critical component of decision management systems. It is a software system designed to process and apply rules, which are logical statements that dictate how decisions should be made.

Imagine you have a set of instructions that tell you what to do under certain conditions—like a recipe.

A Rule Engine works in a similar way but for software. It's a system that applies specific rules to particular data to make complex decisions easier.

With a Rule Engine, businesses can set up, change, and manage the business rules they need to operate without having to rewrite their software code every time something changes.

Components of a Rule Engine



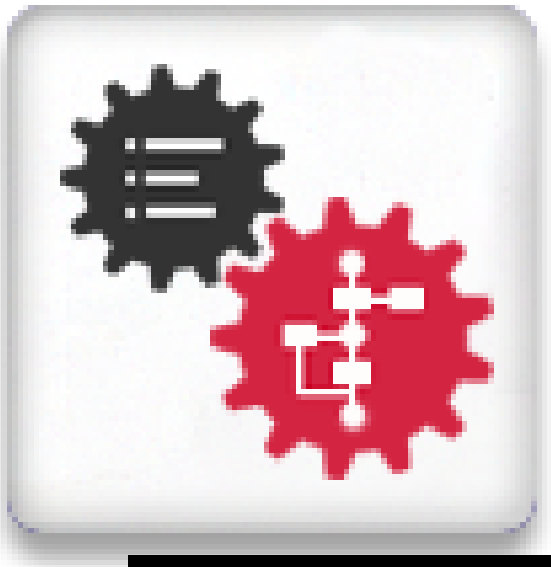
Why Use a Rule Engine?

Flexibility and Efficiency

- Quick Adaptation: Business rules can be updated without altering the core application.
- Consistency: Ensures the same set of rules are applied across various scenarios.
- Transparency: Allows easy understanding and managing of business decisions.
- Scalability: Can handle a growing amount of data and complex decision-making.
- Focus: Developers can concentrate on business logic rather than coding conditions and rules manually.

Remember, a Rule Engine helps separate the 'what' from the 'how' in business logic, making changes and management of the rules something that can often be handled by non-technical users.

Rule Engines: A Brief History



Where It All Began

Go back a few decades and you'll find the beginnings of Rule Engines in academic research and early computing practices. It was all about finding ways to make computers not just calculate numbers but make decisions like a human expert would. This gave rise to "expert systems," a special kind of program that used rules to mimic the decision-making ability of a human expert in specific fields.

From Expert to Engine

The Growth of Automated Decision-Making	Rule Engines evolved from the idea of expert systems but with a broader focus.
	They became more flexible to work with different types of business logic.
	Advancements in AI and computing power made Rule Engines faster and smarter.
	Over time, they moved from niche applications to wider enterprise adoption.
	Integration with other technologies widened their application possibilities.

The Modern Rule Engine

Keeping Up with Business Speed

Today's Rule Engines are all about speed and fluidity in business. They must adapt quickly to changing regulations, market conditions, and business strategies. As such, Rule Engines have become more user-friendly, featuring interfaces that allow business analysts, rather than just developers, to manage rules. This democratization of rule management has made Rule Engines a vital tool in the fast-paced digital world, reinforcing their relevance and necessity for businesses aiming to keep their competitive edge sharp.

redhat

Red Hat

Meet Drools

The Red Hat Rule Engine

Drools, known for its mascot, the JBoss jBPM process manager, is a cutting-edge Rule Engine written in Java. It allows you to write business rules in almost plain English, making them easy to understand and maintain. It's an open-source project that has been around since the early 2000s and is part of the JBoss Enterprise Middleware Suite.

Features of Drools

Standout Capabilities

- Forward and backward chaining: Supports complex reasoning.
- Rule language (DRL): Makes writing rules straightforward.
- Decision tables and spreadsheets: Offers friendly rule authoring for business users.
- Domain-specific languages (DSL): Allows customization of syntax for specific domains.
- RESTful API and decision services: For easy integration with other systems.

How Drools Works

The Drools Mechanism

Drools operates on the principle of 'if-then' logic, where 'if' represents a condition or fact, and 'then' represents the action taken if the condition is met. Drools has a core component known as the Production Memory, which houses the 'if-then' rules. Another key component is the Working Memory, where facts are stored. When Drools is run, it matches facts against the Production Memory to determine which rules are applicable, using a Pattern Matching algorithm known as the Rete algorithm, optimized in Drools as the "ReteOO" network.

Drools in Today's Market

Why Choose Drools?

- Open Source: Free to use, with customizable code
- Flexibility: Adapts to various business requirements and domains.
- Performance: Efficient in processing large numbers of rules and facts.
- Integration: Works well with other Java EE and middleware platforms.
- Community: Backed by a strong community and enterprise support.
- Innovation: Continuously evolving with contributions from around the world.

Conclusion: Drools is a robust Rule Engine and de facto choice for many developers and business analysts due to its capacity to simplify the creation and management of business rules while integrating seamlessly with modern software infrastructures.

AI Meets Rule Engines

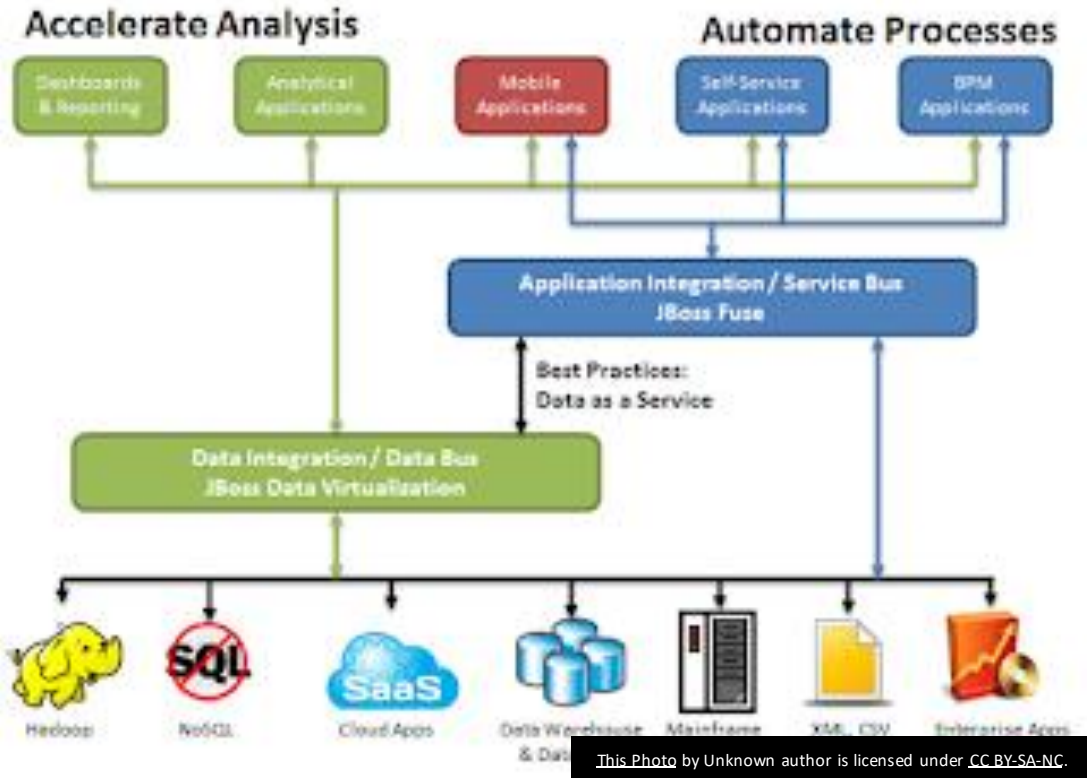
The Convergence of AI and Rule Engines for Smarter Decisions

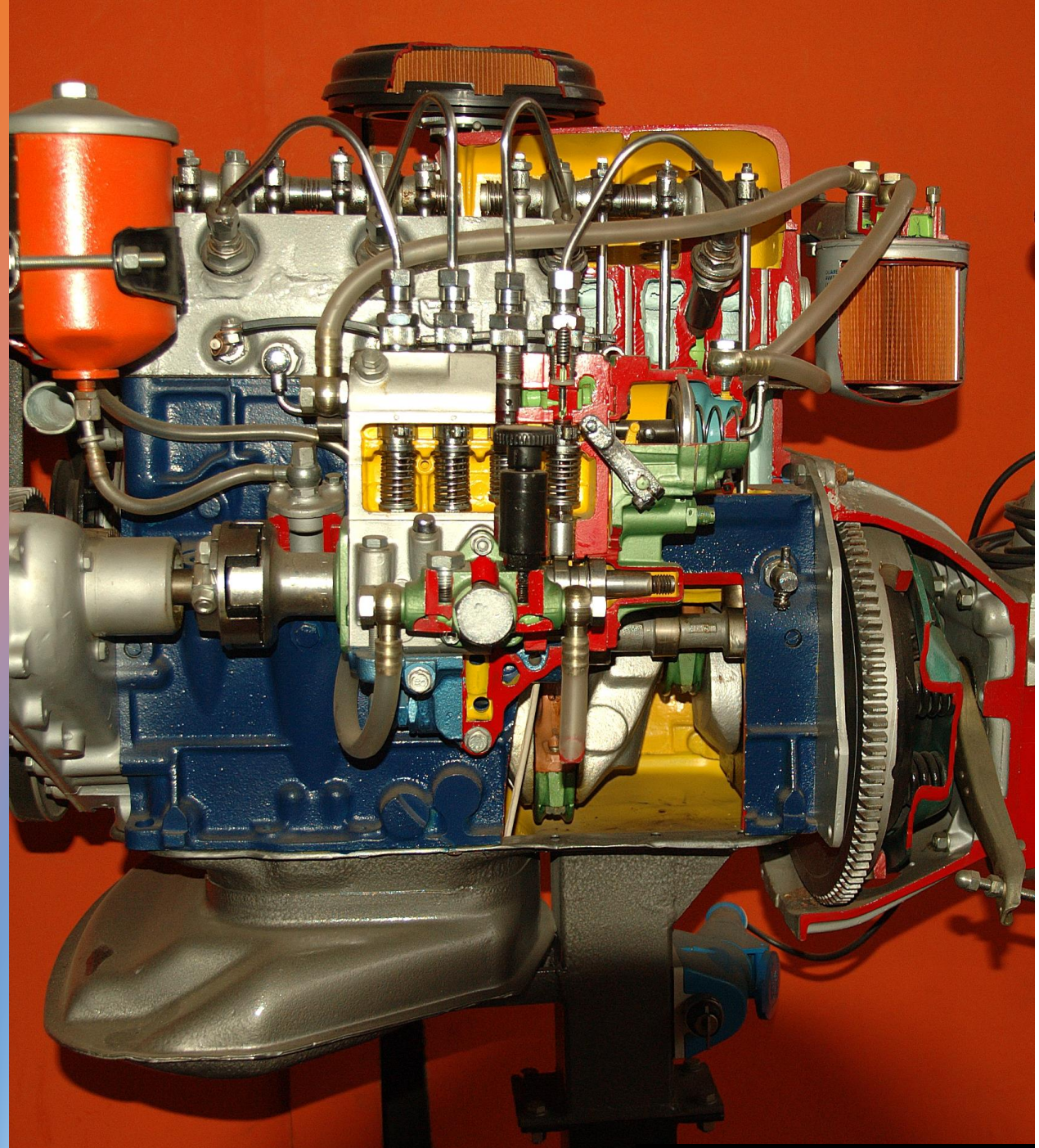
The fusion of Artificial Intelligence (AI) and Rule Engines is a game-changer for decision-making systems. AI brings the power of learning and pattern recognition, while Rule Engines offer a structured approach to execute defined rules. This synergy allows Rule Engines to not only follow prescribed rules but also adapt and learn from new situations, enhancing their ability to respond to complex, real-world scenarios.

Learning Capabilities of Rule Engines

Learning and Adaptation

- Dynamic Rule Adjustment: Rule Engines use AI to modify and create new rules from learned data.
- Predictive Analytics: By integrating with AI, Rule Engines can anticipate outcomes and take pre-emptive actions.
- Natural Language Processing: AI allows Rule Engines to interpret and process rules written in natural language.





The Benefits of AI-Powered Rule Engines

Taking Efficiency to the Next Level

- Incorporating AI into Rule Engines opens up new possibilities, such as automating complex decision processes that would require extensive manual intervention. This leads to increased efficiency, reduced errors, and better scalability as the system can handle an evolving set of rules and conditions. Moreover, the AI component helps in uncovering insights and patterns that can refine the decision-making process and support more informed strategic decisions.

- Conclusion: AI and Rule Engines are a potent combination, providing systems that not only follow business logic but can also evolve and improve over time. This fusion is critical in industries where regulations change frequently or where customer behavior needs to be anticipated and acted upon swiftly.

Setting Up for Drools

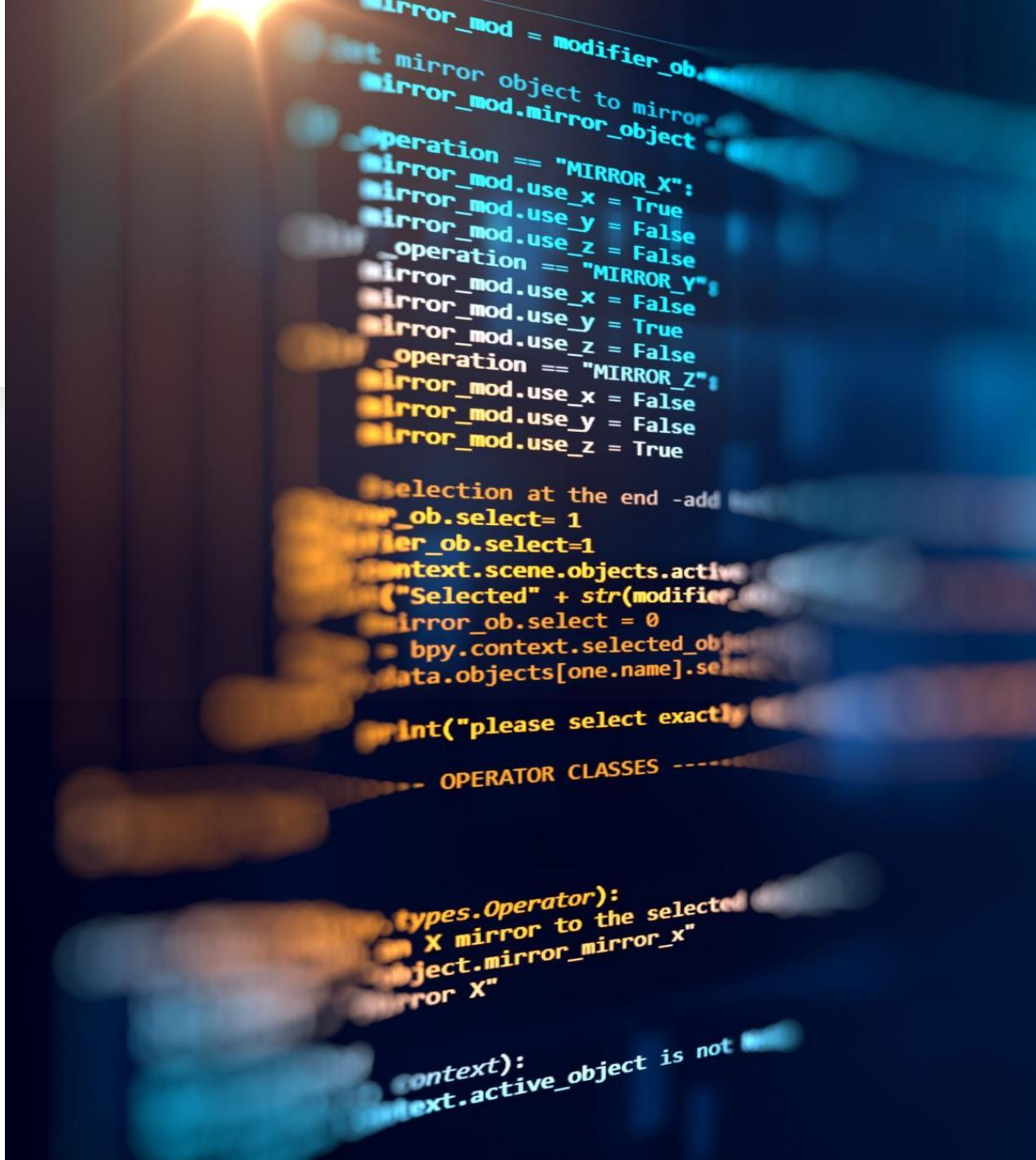
Preparing Your Development Environment

To begin developing with Drools, you need to set up an environment conducive to writing, testing, and deploying rules. This setup typically includes installing Java Development Kit (JDK), a suitable Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA, and the Drools plugin or workbench. Ensuring these tools are correctly configured will facilitate a smooth development experience.

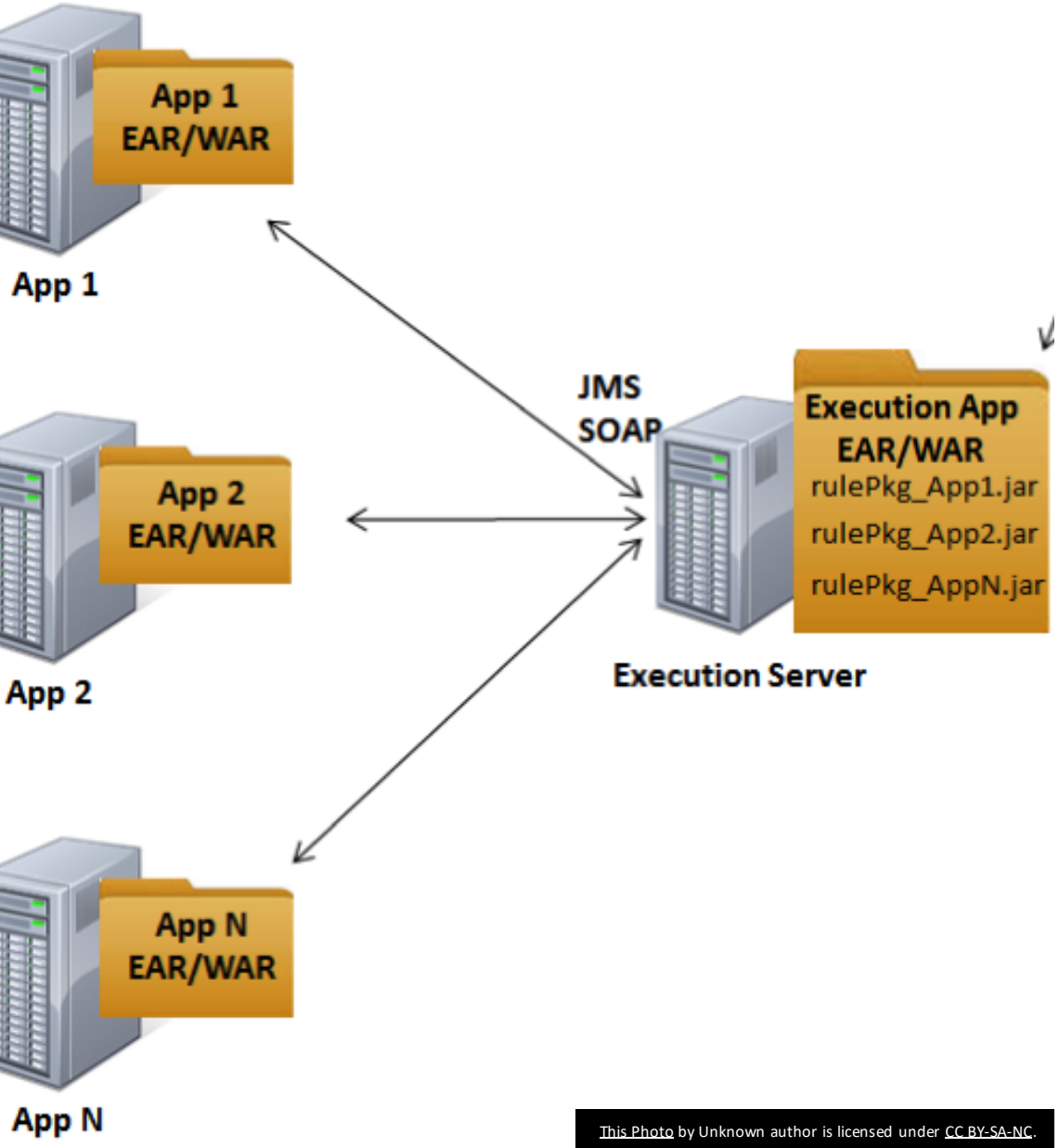
We will walk through the installation process in Module 02

Steps:

1. **Install JDK**: Download and install the latest version of the JDK from Oracle's website (Java Downloads | Oracle)
2. **Set up IDE**: Choose and install your preferred IDE, such as Eclipse or IntelliJ IDEA (Eclipse Downloads | The Eclipse Foundation)
3. **Install Drools Plugin**: For Eclipse, navigate to **Help** -> **Eclipse Marketplace**, search for "Drools", and install the Drools plugin.



Setting Up for Drools



Drools Installation Essentials
The next step is installing Drools and configuring it within your IDE. This involves setting up the Drools plugin or workbench and ensuring that the Drools runtime is properly linked within your project.

Demo:
1. **Download Drools Workbench**: Get the Drools workbench from the Drools Official Website (Drools – Download)
2. **Configure Drools in IDE**: In Eclipse, create a new project and navigate to **Project Properties** -> **Drools**. Add the Drools library to your project

Code:

```
<dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-core</artifactId>
    <version>{latest_version}</version>
</dependency>
```

Replace `**{latest_version}**` with the current version of Drools.

Creating Your First Drools Project

Crafting Your First Rule

Once your development environment is set up, you're ready to create your first Drools project. Start by creating a new project in your IDE and configuring it to use the Drools library. Then, define your data models, write your first business rules in a DRL file, and create a simple application to fire those rules. Testing your rules in the IDE will give you a sense of how Drools processes information and responds to various conditions.

We will get hands-on with Rule Creation in the following Module

Creating Your First Drools Project

Crafting Your First Rule

Steps:

1. **Create New Drools Project**: In Eclipse, select **File** -> **New** -> **Project** -> **Drools Project**.
2. **Define Data Models**: Create Java classes that represent the data you will use in your rules.
3. **Write Rules**: Create a new DRL file in the **src/main/resources** directory and write your first rules.

Code Example:

```
// Define a simple data model
public class Order {
    private String item;
    private int quantity;

    // getters and setters
}

// Sample rule in a DRL file
rule "High Quantity Order"
when
    $order : Order(quantity > 10)
then
    System.out.println("Order of high quantity: " + $order.getItem());
end
```

This rule triggers when an order of more than 10 items is encountered.

Case Studies: Drools in Action

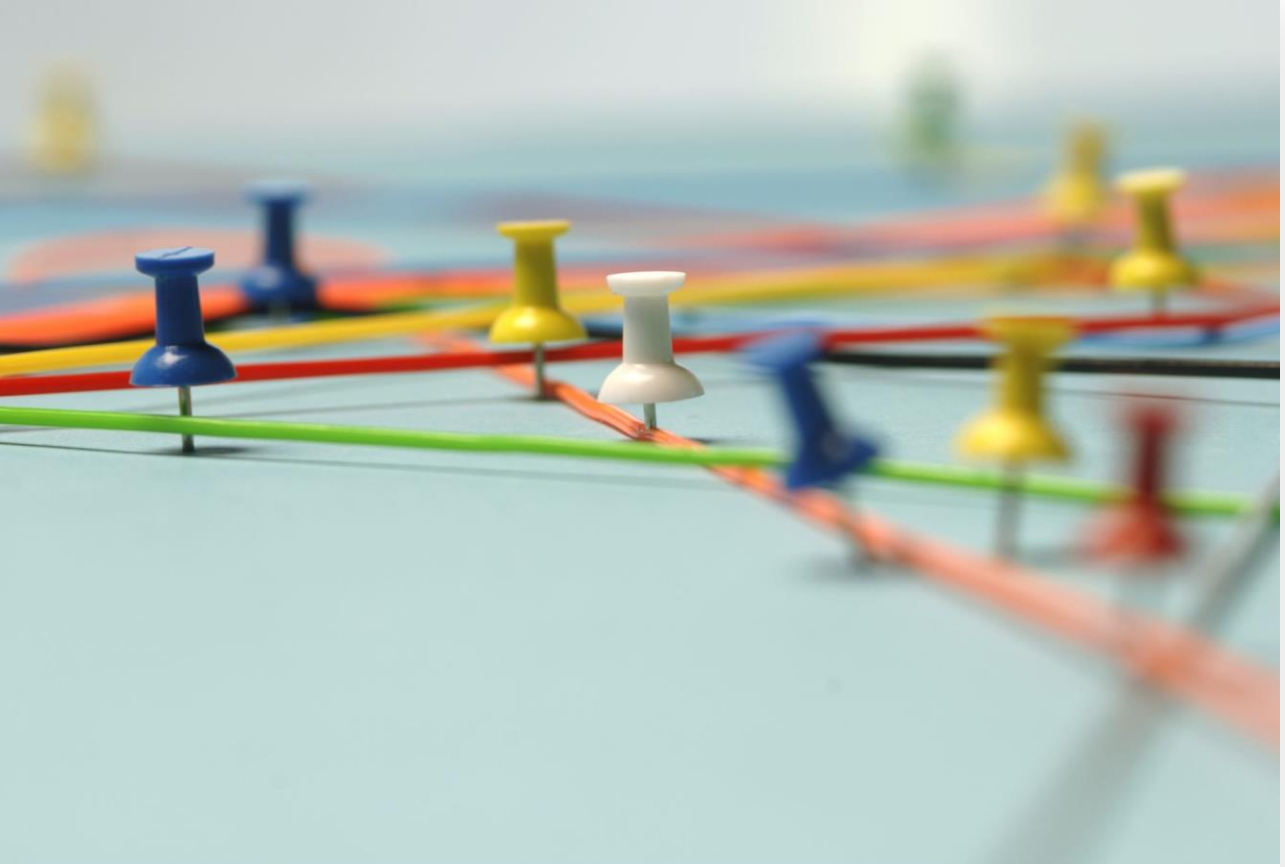
Drools Success Stories

Case studies show how Drools has been successfully implemented across various industries. From healthcare, where Drools manages patient data and treatment protocols, to finance, where it automates loan approval processes, these real-world applications demonstrate how rule engines can optimize operations and decision-making. Each case provides insights into the flexibility and power of Drools when dealing with complex business logic.

Drools Across Sectors

Diverse Industries, One Solution

- Finance: Risk assessment, compliance, and automated decisioning.
- Healthcare: Patient care pathways and clinical decision support.
- Retail: Dynamic pricing, inventory management, and personalized promotions.
- Telecommunications: Fraud detection, service provisioning, and customer retention strategies.



Practical Activities Overview

Module 01 Conclusion

We've reached the end of Module 1, where we've journeyed through the workings of Rule Engines and taken our first steps into the world of Drools 8.4. We started by demystifying what Rule Engines are and their historical significance. We uncovered the unique features and advantages of using Drools, recognizing its role in today's fast-moving business environment. By understanding AI's impact on Rule Engines and taking a tour of real-world case studies, we've seen how powerful Drools can be when it comes to automating complex decisions.

Finally, by setting up our development environments and running practical activities, we've begun to unlock the potential of Drools as a tool for innovation and efficiency. I hope this module has sparked your interest and prepared you for the exciting learning journey ahead. Stay engaged as we continue to explore Drools and its capabilities in the upcoming modules.

Summary

Key Takeaways from Module 1:

- Rule Engines automate decision-making by applying rules to data, providing consistency and transparency in business operations.
- Drools is a powerful Java-based Rule Engine that offers a rich set of features, including a user-friendly rule language (DRL), decision tables, and a strong community.
- The evolution from expert systems to modern Rule Engines reflects the growing need for dynamic business logic management.
- AI enhances Rule Engines by introducing learning capabilities, predictive analytics, and adaptability.
- Setting up a Drools development environment is the first step towards creating efficient rule-based applications.
- Real-world case studies across various industries illustrate the versatility and utility of Drools in addressing complex problems.
- Practical activities help bridge the gap between theoretical knowledge and actual implementation, ensuring a deeper understanding of Drools.

Disclaimer

The information provided in this course, including all the provided documentation, code examples, and case studies, is for educational purposes only. The creators of the course make no representations or warranties of any kind, express or implied, about the completeness, accuracy, reliability, suitability, or availability with respect to the course content or the information, products, services, or related graphics contained within the course for any purpose. Any reliance you place on such information is therefore strictly at your own risk. In no event will the creators be liable for any loss or damage including without limitation, indirect or consequential loss or damage, or any loss or damage whatsoever arising from loss of data or profits arising out of, or in connection with, the use of this course material.

Copyright Information

TBF

DROOLS 8

MODULE 02:
Getting Started with Drools

Laying the Groundwork for Rule Management

Presented by John Paul Franke

Course Overview

In Module 2, we'll dive into the hands-on aspects of Drools 8.4. We're going to guide you through the initial stages of creating and managing a Drools project. Our journey will take us through setting up your Drools environment, familiarizing ourselves with the Drools project structure, and getting our hands dirty by writing some simple rules and decision tables. This practical introduction to Drools will set the stage for you to effectively use this powerful rule engine to automate complex decisions within your or your clients' businesses

Learning Objectives

In Module 02, we will:

- Verify and test a Drools environment and understand project structure.
- Gain hands-on experience in writing and testing simple Drools rules.
- Learn the basics of Decision Model and Notation (DMN) and its application in decision management.
- Develop practical knowledge by creating a DMN-based traffic violation decision service.
- Identify best practices and common pitfalls in rule creation and management with Drools and DMN.

Setting Up Your Drools Environment

To kick off your foray into Drools development, you'll need to start with setting up the right environment. This includes having the Java Development Kit (JDK) installed and choosing a suitable Integrated Development Environment (IDE) that supports Drools, like Eclipse, IntelliJ IDEA, or VSCode with appropriate Drools extensions. This setup will enable you to write, compile, and test your Drools applications effectively.

Fine-Tuning Your Drools Setup

- **Install Drools plugins or workbench:** These tools integrate Drools functionalities directly into your IDE.
- **Configure the Drools runtime:** Ensure that the runtime is properly linked and recognized within your development environment.
- **Set up a Maven project:** Use Maven for dependency management of Drools libraries to streamline your project setup.

Verifying Your Setup

Once the Drools environment is installed and configured, it’s crucial to verify that everything is working correctly. You can do this by creating a 'Hello World'-style rule within your IDE and running it to ensure that the setup is correct. This initial test helps identify and fix any configuration issues early, laying a solid foundation for future development with Drools.

Verify Your Setup

Setting Up a Simple Rule

- Open your IDE and navigate to the Drools project.
- Create a new DRL file in the `src/main/resources` directory.
- Write the following simple rule:

```
package com.example.rules

rule "Hello World"

When

    eval(true)

Then

    System.out.println("Hello, World!");

End
```

Use Maven: `mvn clean install` to build the project.

Testing Your Rule Environment

Confirming Rule Execution

Steps:

- Create a new Java class in `src/main/java`.
- Include necessary imports (`org.kie.api.KieServices`, `org.kie.api.runtime.KieContainer`, `org.kie.api.runtime.KieSession`).
- Initialize the Drools environment and create a `KieSession`:

```
KieServices ks = KieServices.Factory.get();

KieContainer kc = ks.getKieClasspathContainer();

KieSession kSession = kc.newKieSession();
```

Executing and Observing the Rule

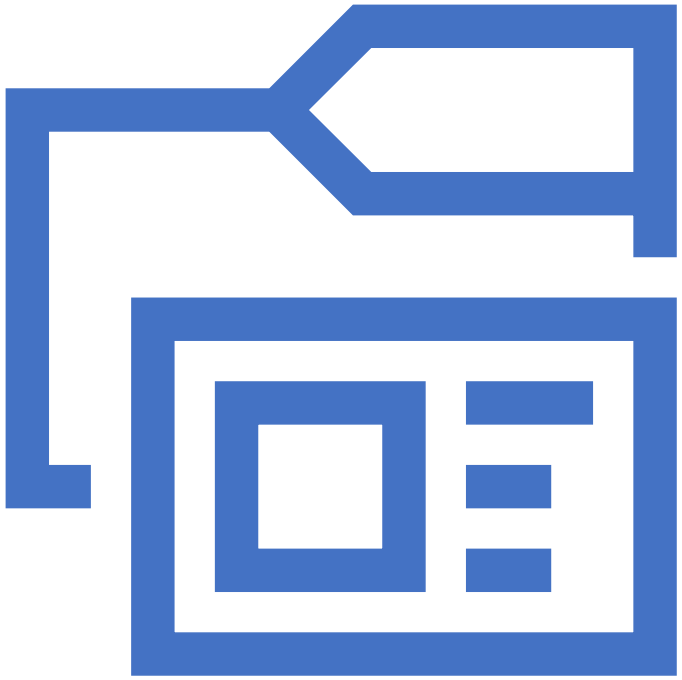
Steps:

- Add `kSession.fireAllRules();` to trigger rule execution.
- Check the console for the "Hello, World!" output.

Rule Files and Resources

Rule Files and Supporting Artifacts

Within the Drools project structure, rule files (.drl) are the heart where the business logic lives. Alongside these are other resources like decision tables (.xls or .csv for spreadsheet-based rules) and potential domain-specific language definitions (.dsl). These resources work in concert, defining rules that Drools uses to evaluate and make decisions, which is why it’s crucial to understand how each piece fits into the project puzzle.



The Drools Directory Layout

Inside a Drools Project

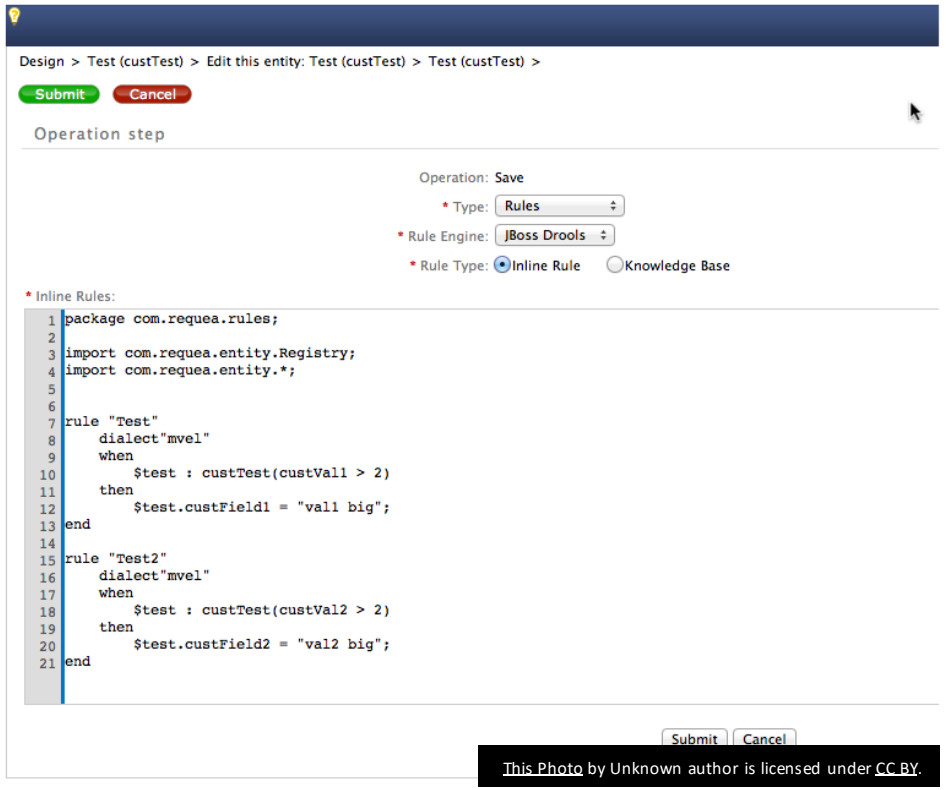
Drools projects maintain a specific directory structure that lays the groundwork for organized rule management. Key components include the 'src/main/resources' folder, where your Drools rule files (.drl) and other resources are placed. Understanding where to store and how to reference these components is critical for ensuring your project runs smoothly.

the Drools Project Structure

Package Structure and Namespaces

In a Drools project, rules and resources are often grouped into packages, similar to how Java classes are organized. This packaging structure promotes good organization and makes it easier to manage and reference different sets of rules, especially in larger projects. Namespaces, indicated within the rule files, prevent conflicts and ensure rules are executed within the appropriate context. Adopting a consistent and logical project structure is key to maintaining a clean and efficient rule base as your Drools applications grow.

Crafting a Drools Rule



Writing Your First Rule

Creating your first rule in Drools is your initiation into the world of business rule management. A simple 'Hello World' rule is the perfect starting point. In Drools, rules are written in a .drl file, where you define when a certain condition is met (the "when" part) and what action should be taken (the "then" part). This basic structure will lay the foundation for more complex rules that you'll develop as you progress through the course.

Rule Syntax and Structure

DRL Grammar	
Drools Rule Language (DRL) is designed to be readable and accessible.	"rule": Starts the definition of a rule.
	"when": Specifies the conditions or patterns to match.
	"then": Defines the actions to execute when conditions are met.
	"end": Closes the rule definition.

Rule Syntax and Structure

DRL Grammar

Understanding the syntax and structure is crucial to writing effective rules. Here's a simple rule example:

```
rule "Say Hello"

when

  // Conditions to match

then

  System.out.println("Hello,  World!");

end
```

Testing and Debugging Rules

Test Runs and Fine-tuning

After writing your first rule, the next step is to test it within your Drools environment. This involves creating a session, inserting facts into it, and firing the rules to observe the outcome. Testing helps validate the correctness of your rules and is a great opportunity to get familiar with the debugging tools and techniques that will assist you in troubleshooting and refining your rules to ensure they perform as intended.

See Lab **'Testing and Debugging Rules in Drools'**

Decision Model and Notation (DMN)

Deciphering DMN

DMN, or Decision Model and Notation, is an open standard managed by the Object Management Group (OMG). It provides a common notation that's understandable by all business users, from the analysts who create decision requirements to the technical developers responsible for implementing them. This shared framework helps organizations map out, standardize, and communicate complex decision logic with clarity and precision.

DMN Elements

The Components of DMN

- Decision: A conclusion reached after evaluating defined criteria.
- Input Data: Information required to determine or influence a decision.
- Knowledge Source: Authority or reference that underpins a decision.
- Business Knowledge Model: Encapsulated logic that can be reused across decisions.

DMN in Drools

Integrating Standards with Rule Engine Technology

Drools provides an execution engine for DMN models, allowing you to include DMN decision requirements directly within your rule projects. This enables you to blend the structured logic of DMN with the dynamic rule evaluation capabilities of Drools. Implementing DMN within Drools means decisions can be managed more formally with a clear representation that's both business-friendly and execution-ready.

Practical Exercise: Traffic Violation Decision Service

Preparation

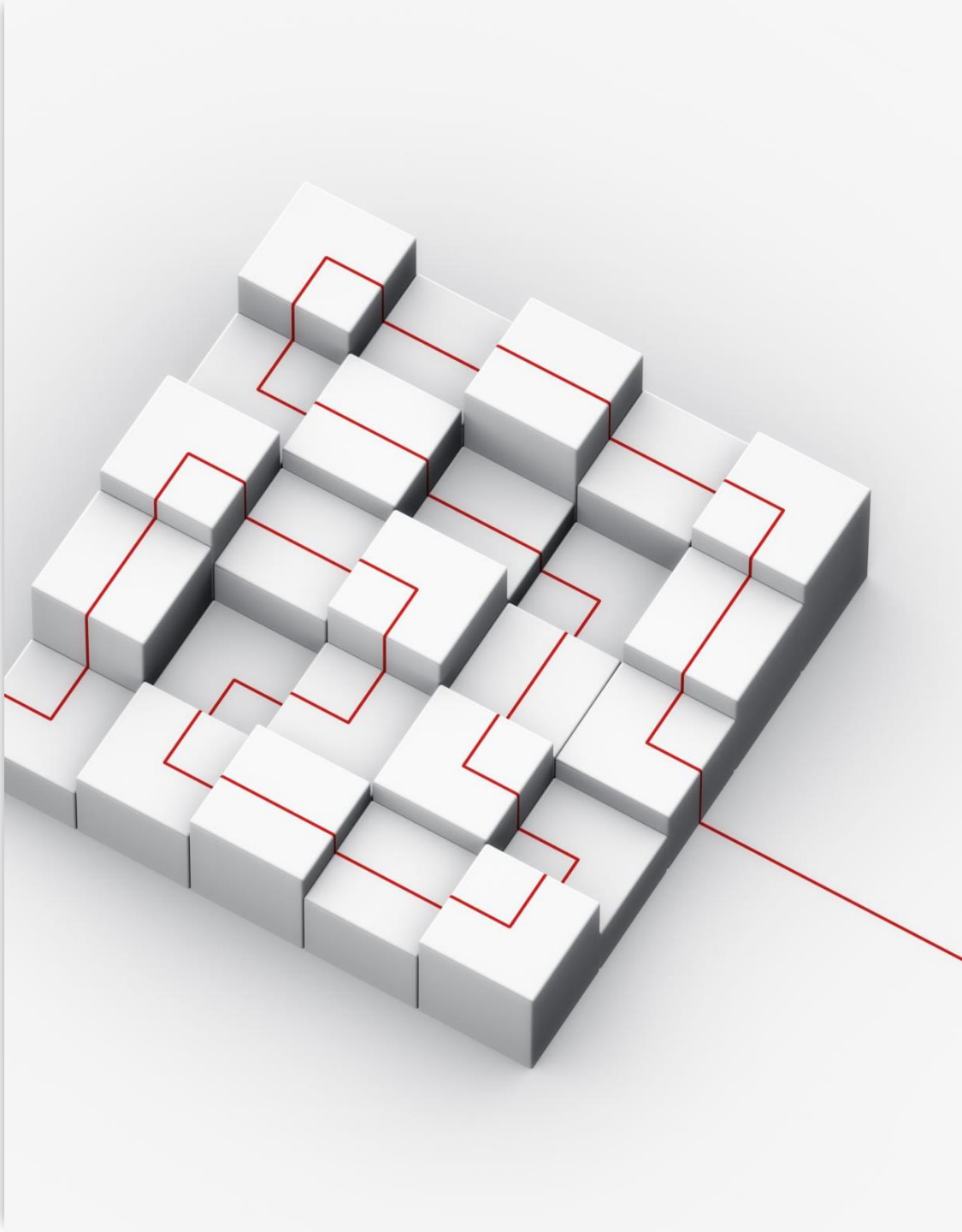
To apply what we've learned about DMN and Drools, we'll tackle a practical exercise: constructing a decision service for traffic violation management. This scenario is designed to test our understanding of the DMN components and how they interact within a Drools application. We'll start by outlining the requirements and defining the input data, decisions, and business knowledge models that form the basis of our traffic violation decision logic.

Constructing the Decision Logic

Building the DMN Model

In this step of the exercise, you will use Drools to visually construct your DMN model. By mapping out the relationships between decisions, input data, and business knowledge models, you will create a comprehensive representation of the traffic violation decision-making process.

- Identify the decisions to be made, such as calculating fines and determining driving suspension.
- Define the input data needed, like the speed limit, actual speed, and driver's violation history.
- Develop the business knowledge models that encapsulate the logic for decision-making.
- Use Drools DMN authoring tools to visually represent the decision requirements graph.



Exercise Review

Consolidating Knowledge Through Practice

As we conclude our traffic violation decision service exercise, it's time to review and reflect on what we've learned. Consider how best practices influenced the effectiveness of your decision service and how anticipating pitfalls helped you create a more resilient DMN model. These insights will be invaluable as you continue to develop and refine decision services using Drools and DMN in your projects.

Quiz Time!

Quiz Questions and Answers:

1. What JVM language is primarily used for writing Drools rules?
   - A. Python
   - B. Scala
   - C. Java
   - D. Ruby
   - Correct Answer: C
2. Which file extension is associated with Drools rule files?
   - A. .dsl
   - B. .java
   - C. .drl
   - D. .xml
   - Correct Answer: C
3. In DMN, what diagram represents the decision logic and its dependencies?
   - A. Class Diagram
   - B. ER Diagram
   - C. Decision Requirement Graph (DRG)
   - D. Use Case Diagram
   - Correct Answer: C

Quiz Time!

Quiz Questions and Answers:

1. What should you consider to ensure the correct execution order of rules in Drools?
   - A. The size of the DRL file
   - B. Salience attribute
   - C. The timestamp of rule creation
   - D. The alphabetical order of rule names
   - Correct Answer: B

2. Where do you typically store your Drools rule files in a Maven project?
   - A. src/main/java
   - B. src/main/resources
   - C. src/main/scripts
   - D. src/main/webapp
   - Correct Answer: B

DROOLS 8

MODULE 03:
BUILDING, DEPLOYING, UTILIZING, & RUNNING DROOLS

COMPREHENSIVE GUIDE TO OPERATING DROOLS APPLICATIONS

Presented by John Paul Franke

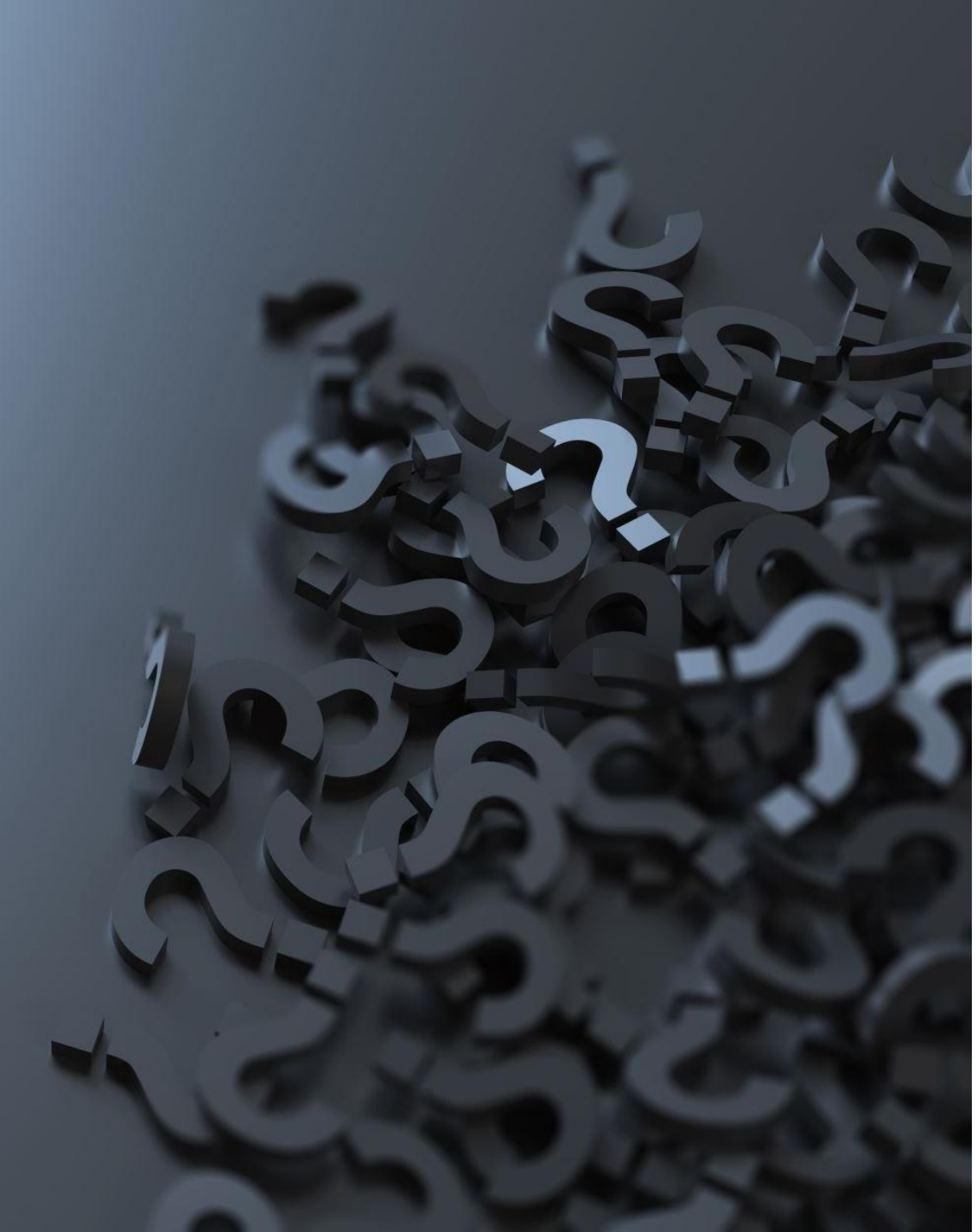
Course Overview

Welcome to Module 3, where our focus turns to the practical aspects of building, deploying, utilizing, and running Drools applications. Throughout this module, we'll gain valuable experience in constructing a Drools project from the ground up, explore strategies for deploying applications, and learn how to run them effectively. We'll also dive into dynamic rule management with the KieScanner and discover best practices for implementing persistence and handling transactions within Drools. By engaging in a practical exercise, you'll solidify your understanding by building and deploying a sample application. This module is all about turning your Drools knowledge into actionable skills that can be applied in real-world scenarios.

Learning Objectives

This Module's Goals

- Understand the steps involved in building a Drools project, including rule authoring and testing.
- Learn how to deploy Drools applications across different environments.
- Explore how to run Drools applications and manage the execution of rules.
- Discover the use of KieScanner for dynamic updates in rule management.
- Dive into the concepts of persistence and transactions within Drools.
- Apply the learned concepts by building and deploying a sample Drools application.
- Prepare for the hands-on exercises that will reinforce your Drools knowledge.
- Get ready to showcase your understanding of the module's content through the assessment.



Building a Drools Project (v1)

Creating the Rule Base

The first step in building a Drools project is to establish a solid rule base. This involves writing your business rules using business-friendly languages, such as Drools Rule Language (DRL) or decision tables. It’s about translating complex business logic into rules that the Drools engine can understand and execute, which requires a combination of technical attention to detail and a strong understanding of the underlying business processes.

Building a Drools Project (v2)

Creating the Rule Base

The first step in building a Drools project is to establish a solid rule base. This involves writing business rules using Drools Rule Language (DRL). It's about turning complex business logic into executable rules that the Drools engine can process.

Demo:

```
// Sample rule in a .drl file

rule "Discount for Preferred Customer"

when

   $customer : Customer(type == Customer.Type.PREFERRED)

   $order : Order(customer == $customer, orderTotal >= 100)

then

   $order.setDiscount(10);

   update($order);

end
```

Configuring the Project (v1)

Configurations and Dependencies

- Define the project's Maven or Gradle build configuration for handling dependencies and plugins.
- Set up the KIE (Knowledge Is Everything) module by configuring the kmodule.xml file with the necessary KIE bases and KIE sessions.
- Manage external facts and data models that the rules will operate on by establishing them within the project's structure.

Configuring the Project (v2)

Configurations and Dependencies

Once you have the rule files, configure your project's build definition to handle dependencies and plugins for Drools.

Demo:

```
<!-- Maven pom.xml snippet for Drools dependency -->

<dependencies>

  <dependency>

    <groupId>org.kie</groupId>

    <artifactId>kie-api</artifactId>

    <version>${drools.version}</version>

  </dependency>

  <!-- other dependencies -->

</dependencies>
```

Rules Compilation and Packaging

Wrapping Up the Project Build

After defining and organizing your rule files, the next phase is compiling them into a deployable package, such as a JAR file. This compilation process involves converting the human-readable DRL files into a binary format that the Drools engine can process.

Demo:

Maven commands to compile and package a Drools project:

- mvn clean compile
- mvn package

The Deployment Process

Deployment Overview

Deployment is the phase where your Drools application moves from a development setting to a production environment. Key steps include packaging your compiled rules and application code into a deployable artifact, choosing a suitable deployment platform (like a local server, cloud service, or containerization), and managing configuration settings for different environments.

Maven command to deploy the package to an artifact repository:

- mvn deploy

Choosing a Deployment Platform

The choice of deployment platform should align with your application's requirements, such as scale, load, and integration with other services. Consider factors like ease of management, security, and the expertise available within your team.

- Local Server: Good for initial testing and small-scale applications.
- Cloud Services: Offers scalability and high availability for enterprise-level solutions.
- Containers (Docker/Kubernetes): Provides portability and consistency across environments.
- Application Servers (WildFly, Tomcat): For traditional Java EE application deployments.

 # An example Dockerfile line for deploying a Java application

```
FROM openjdk:11-jre-slim

COPY target/drools-app.jar /usr/app/

WORKDIR /usr/app

CMD ["java", "-jar", "drools-app.jar"]
```

Deploying with Maven and Jenkins

Automating the Deployment

For a smooth and reliable deployment process, automation is key. Using tools like Maven and Jenkins, you can set up Continuous Integration/Continuous Deployment (CI/CD) pipelines to automate the building, testing, and deployment phases. This setup ensures that changes to your rules and application code are consistently integrated and pushed to production with minimal manual intervention. The pipeline will handle compiling the rules, running tests, packaging the artifacts, and, finally, deploying them to the chosen environment.

Deploying with Maven and Jenkins

- **Automating the Deployment**

- Demo: (This slide is a mess and needs structuring… somehow)

```
// Jenkins pipeline script snippet for deploying Drools project
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        script {
          // Building the Drools project
          sh 'mvn clean package'
        }
      }
    }
    stage('Deploy') {
      steps {
        script {
          // Deploying the package
          sh 'mvn deploy'
        }
      }
    }
  }
}
```

Initiating the Drools Runtime

Launching Rule Execution

To run your deployed Drools application, you start the Drools runtime, which is the engine responsible for executing the rules. This involves initializing a KIE container with the packaged KJAR (Knowledge JAR) that contains your rules and setting up the KIE session. A KIE session is where data, known as facts, are inserted, and where the rules have their logic applied. This session can be stateful—retaining knowledge across invocations—or stateless for one-time executions.

Initiating the Drools Runtime

Demo:

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();
KieSession kSession = kContainer.newKieSession("ksession-rules");

kSession.insert(factObject);
int firedRules = kSession.fireAllRules();

System.out.println("Number of Rules executed = " + firedRules);
kSession.dispose();
```

Real-time Rule Firing and Response

Dynamic Decision-Making in Action

Once the Drools runtime is up and running, the Drools engine continuously listens for new facts to be inserted into the working memory. As facts arrive, they are matched against the rule conditions—when the conditions are met, the corresponding actions are then executed. This ability to fire rules in response to data in real-time is a core benefit of using Drools, facilitating dynamic response in applications like fraud detection, inventory management, or adaptive business workflows.

Real-time Rule Firing and Response

Dynamic Decision-Making in Action

Demo:

```
// Assuming `kSession` has already been created as per the previous slide

MyFact fact = new MyFact();

fact.setProperty("value");


kSession.insert(fact);

kSession.fireAllRules();
```

Monitoring and Managing Rule Execution

Ensuring Efficiency and Correctness

Running a Drools application isn't just about starting the engine and walking away. Active monitoring of rule execution allows for timely identification of performance issues or incorrect rule behavior. Tools such as the Drools Workbench or custom logging can be employed to track which rules are firing, how long they take, and what the outcomes are. Effective management of the rule execution environment ensures your Drools application remains performant and reliable throughout its lifecycle.

Demo:

```
kSession.addEventListener(new DebugAgendaEventListener());

kSession.addEventListener(new DebugRuleRuntimeEventListener());



// Insert facts and fire rules as demonstrated previously
```

Introduction to KieScanner

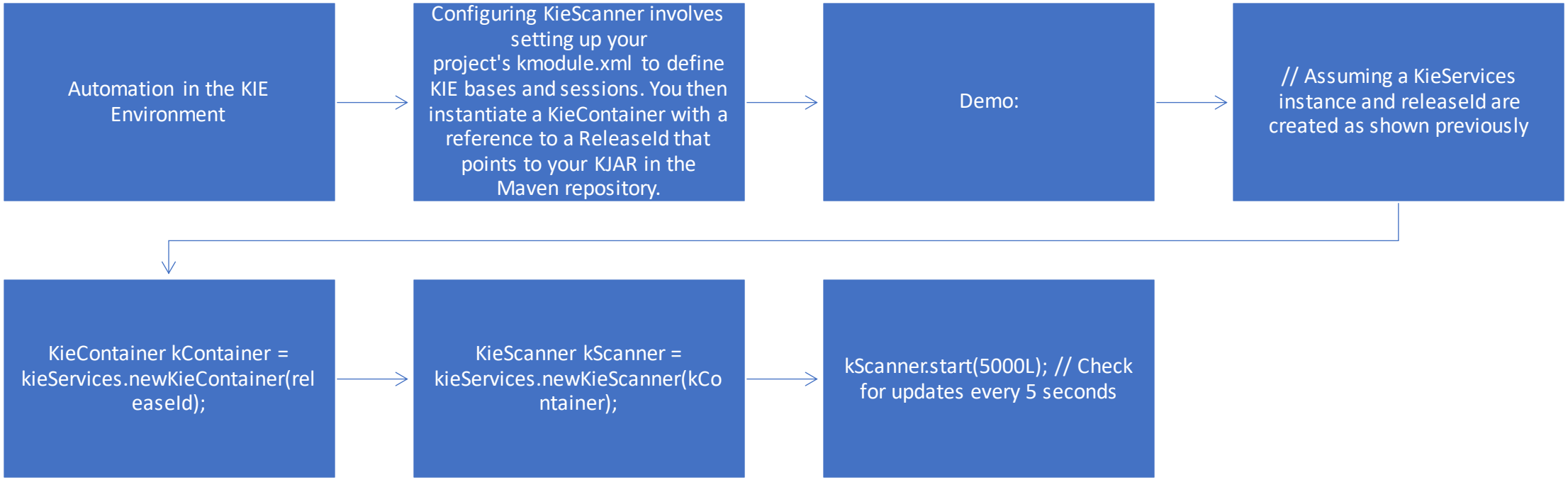
Dynamic Rule Updates with KieScanner

The KieScanner is a tool that enables Drools applications to update the rules at runtime without restarting the application. It polls the Maven repository for updated versions of your KJAR (Knowledge JAR) and, when a new version is found, the KieContainer updates to reflect these changes. This feature supports agility and continuous integration by allowing rule changes to be pushed into production quickly and efficiently.

Demo:

```
KieServices kieServices = KieServices.Factory.get();

ReleaseId releaseId = kieServices.newReleaseId("com.example", "my-drools-kjar", "LATEST");

KieContainer kieContainer = kieServices.newKieContainer(releaseId);

KieScanner kieScanner = kieServices.newKieScanner(kieContainer);

kieScanner.start(10000L); // Scan for new versions every 10 seconds
```

Setting Up KieScanner



Managing KieScanner Updates

Streamlining Rule Management

While KieScanner brings the convenience of updating rules without downtime, it also introduces complexities related to rule versioning and consistency. Proper versioning practices in your Maven setup are critical to prevent disruptions. Additionally, you should implement safeguards to ensure that updates do not break existing rule executions, which may involve automated testing or canary releases.

Through the proper use of KieScanner, you can ensure your Drools application remains up-to-date with the latest business rules, providing the agility needed to respond to changes in business logic rapidly.

Managing KieScanner Updates

Demo:

```
// Monitoring KieScanner updates
KieScanner kScanner = kieServices.newKieScanner(kieContainer);
kScanner.addListener(new KieScannerEventListener() {
   @Override
   public void
onKieScannerStatusChangeEvent(KieScannerStatusChangeEvent
statusChange) {
      // Handle the status change event here
      if (statusChange.getStatus() == KieScannerStatus.UPDATE) {
         System.out.println("A new update has been deployed");
      }
   }
});
kScanner.start(5000L);
```

Persistence and Transactions in Drools

The Role of Persistence

Persistence in Drools allows the state of sessions to be maintained across application restarts and crashes, enabling long-running processes to recover and continue where they left off. This involves persisting facts, rules, and the working memory state into a database. Drools provides integration with JPA (Java Persistence API) for seamless persistence management.

Persistence and Transactions in Drools

Demo

```
KieServices kieServices = KieServices.Factory.get();

KieContainer kContainer =
kieServices.getKieClasspathContainer();

EntityManagerFactory emf =
Persistence.createEntityManagerFactory("org.jbpm.persistence
.jpa");


Environment env = KnowledgeBaseFactory.newEnvironment();

env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);

env.set(EnvironmentName.TRANSACTION_MANAGER,
TransactionManagerServices.getTransactionManager());


StatefulKnowledgeSession kSession =
JPAKnowledgeService.newStatefulKnowledgeSession(kBase,
null, env);
```

Transactions Management

Consistency in Rule Processing

Transactions in Drools ensure that rule executions complete successfully or roll back in entirety in case of errors, maintaining data integrity. Drools can integrate with transaction management frameworks to control transaction boundaries, ensuring that changes to the working memory are committed or reverted atomically.

Transactions Management

Demo

```
UserTransaction transaction = (UserTransaction) new InitialContext().lookup("java:comp/UserTransaction");
transaction.begin();


try {
    kSession.insert(someFact);
    kSession.fireAllRules();


    transaction.commit();
} catch (Exception e) {
    transaction.rollback();
    throw e;
}
```

Combining Persistence and Transactions

Synchronizing

Using persistence and transactions together enhances the robustness of your Drools application, preventing data loss and ensuring that all business rule operations are performed consistently. It is critical to configure both correctly to handle the real-time nature of rule firing while also accounting for the recovery of the session's state in case of discrepancies.

Adopting these approaches guarantees that your Drools applications will handle the full lifecycle of rule processing - from initial firing to state persistence - with the necessary reliability and consistency.

Combining Persistence and Transactions

Demo

```
// Similar to earlier demos, now combined within a transaction context
// and using an EntityManager for JPA persistence

EntityManagerFactory emf =
Persistence.createEntityManagerFactory("org.drools.persistence.jpa");
EntityManager em = emf.createEntityManager();

UserTransaction ut = (UserTransaction) new
InitialContext().lookup("java:comp/UserTransaction");
ut.begin();
em.joinTransaction();

// Create and use the kSession
// ...

em.flush();
ut.commit();
em.close();
```

Practical Exercise:
Building and Deploying an Application

See lab 03 - Building and Deploying an Application

We will consolidate our learning by engaging in a practical exercise to build and deploy a simple Drools application. This exercise will cover from setting up the environment and writing rules, to compiling, packaging, and deploying the application. By the end of this exercise, you will have a functional application that demonstrates how Drools handles decision automation in a real-world context.

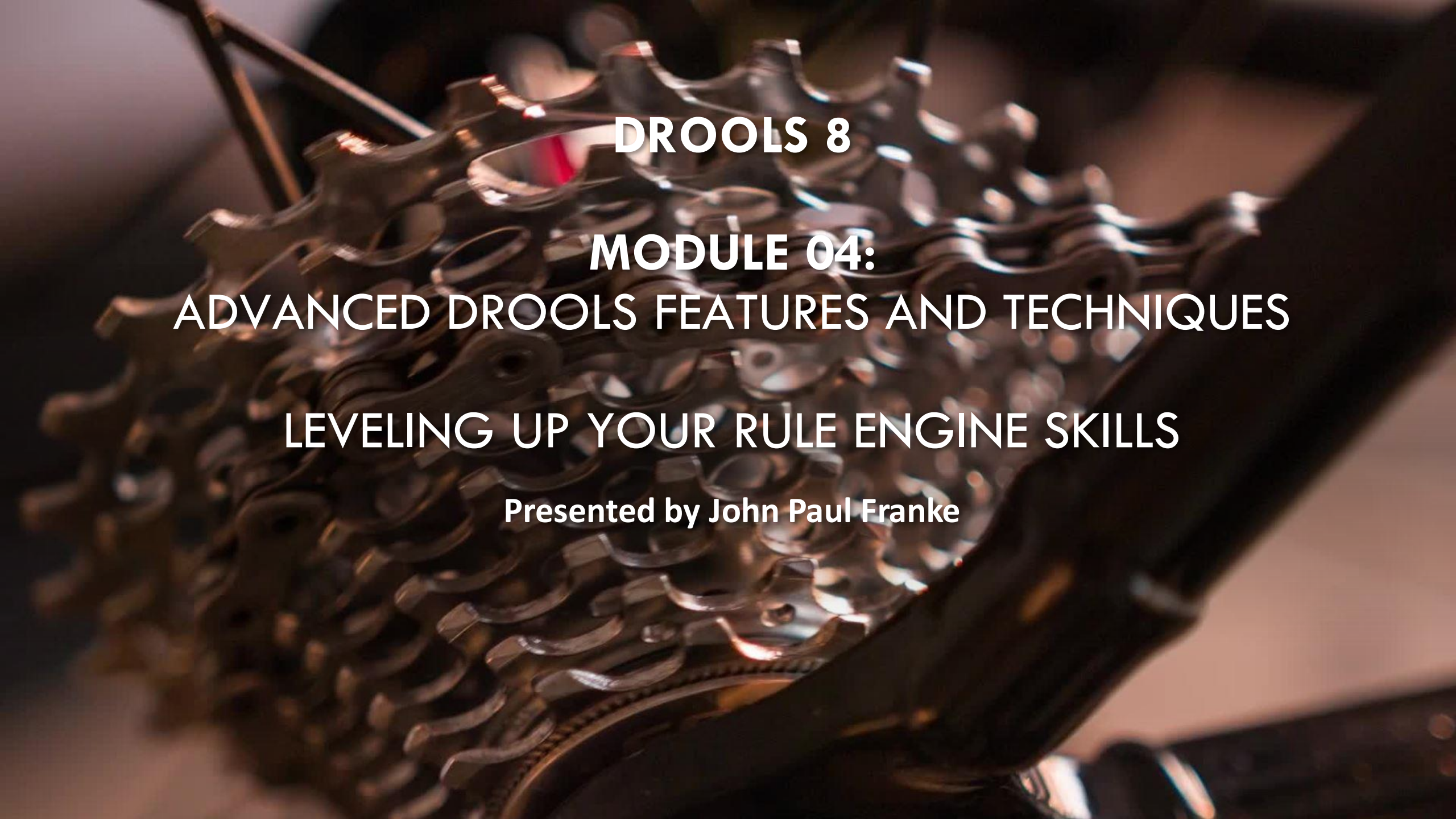
Conclusion

As we conclude Module 03, we've gone over the essentials of building, deploying, utilizing, and running Drools applications. You've seen how a Drools project comes to life from initial setup to dynamic, real-world rule execution. This module provided you with both the theoretical framework and practical experience essential for working with Drools applications in a production environment. With the knowledge of KieScanner, persistence, and transactions, you are now better equipped to create robust and responsive rule-based applications.

Summary

Key Takeaways from Module 03

- We learned the processes involved in building a Drools project, including writing and compiling rules.
- We explored various deployment strategies and platforms, understanding how to bring Drools applications into production.
- We delved into running Drools applications and managing sessions for rule executions.
- We discovered the KieScanner's role in enabling dynamic rule updates without downtime.
- We covered the importance of persistence and transactions in maintaining state and ensuring data integrity within Drools applications.
- We undertook a practical exercise to build and deploy a simple Drools application, cementing our understanding of the end-to-end process.
- We reviewed best practices for rule management and common challenges faced when working with rule engines.



DROOLS 8

MODULE 04:
ADVANCED DROOLS FEATURES AND TECHNIQUES

LEVELING UP YOUR RULE ENGINE SKILLS

Presented by John Paul Franke

Course Overview

Welcome to Module 04 of this course on Drools 8.4, where we will uncover the secrets behind KIE Sessions, understand how Drools manages complex event processing (CEP), and learn about advanced rule execution controls. We will also explore the nuances of fact models, propagation modes, and performance tuning to optimize your Drools applications. By the end of this module, you'll have a thorough understanding of the sophisticated capabilities of Drools, empowering you to build more efficient and intelligent rule-based systems.

Learning Objectives

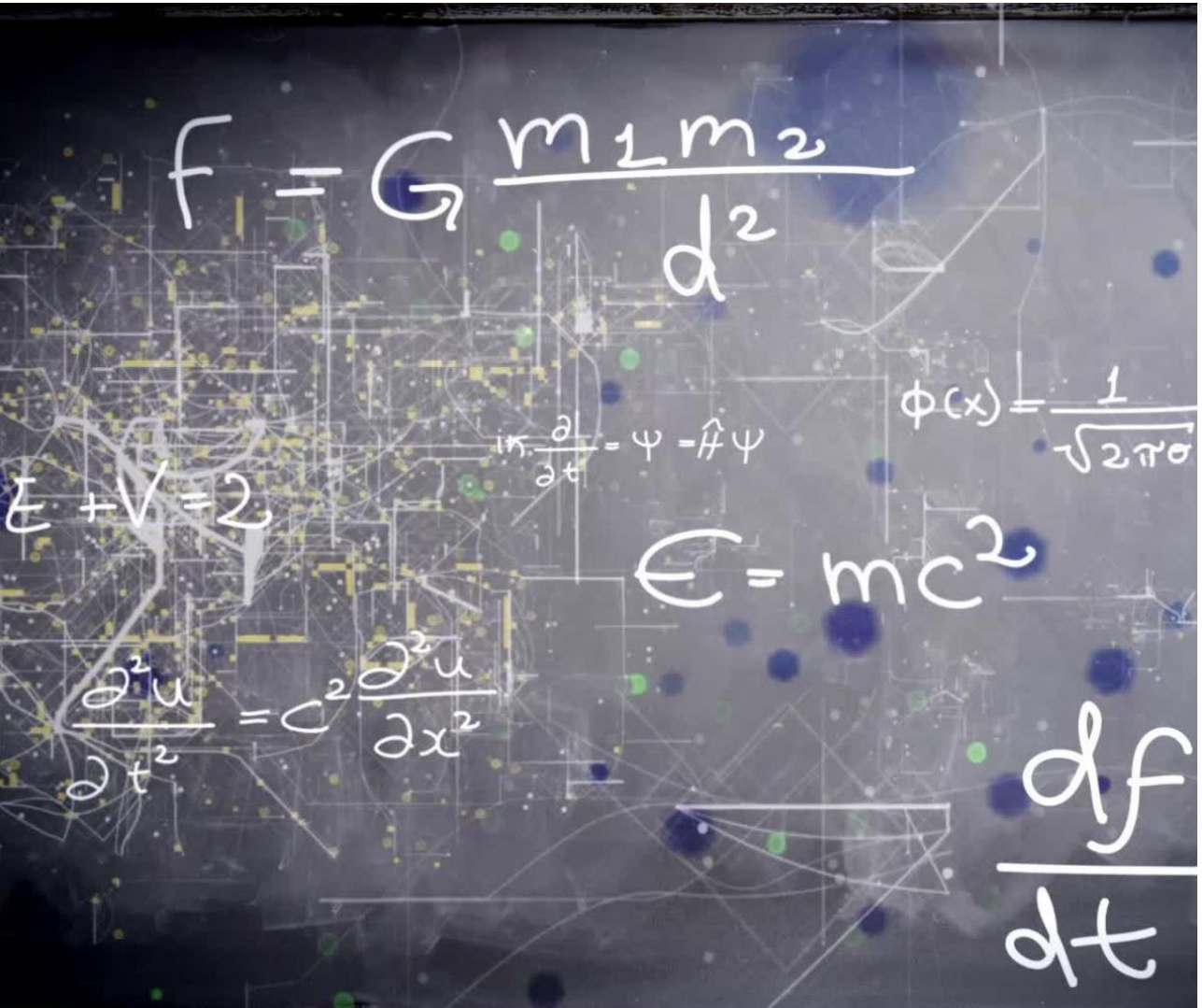
TBD

Understanding KIE Sessions

The Kie Project Concept

A Kie Project follows the structure of a conventional Maven project, only with the unique inclusion of a kmodule.xml file. This file, placed in the resources/META-INF folder, declaratively defines the KieBases and KieSessions that can be instantiated. To compile all the Kie assets beneath the resources folder into the KieBase, you need to create a KieContainer for them.

Understanding KIE Sessions



Demo:

<!-- An empty kmodule.xml file -->

<?xml version="1.0" encoding="UTF-8"?>

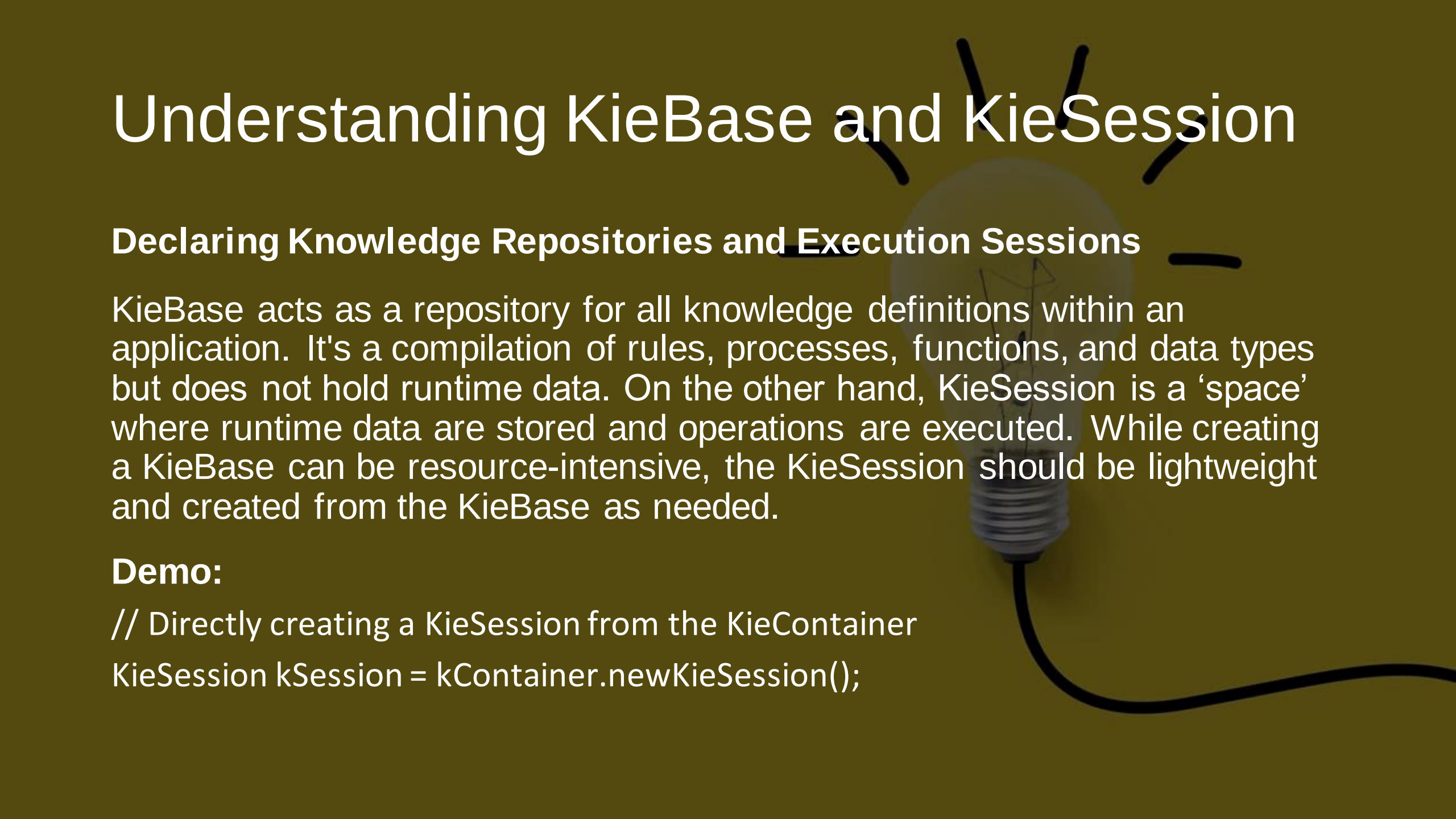
<kmodule xmlns="http://www.drools.org/xsd/kmodule"/>

// Creating a KieContainer from the classpath

KieServices kieServices = KieServices.Factory.get();

KieContainer kContainer = kieServices.getKieClasspathContainer();

Understanding KieBase and KieSession



Declaring Knowledge Repositories and Execution Sessions

KieBase acts as a repository for all knowledge definitions within an application. It's a compilation of rules, processes, functions, and data types but does not hold runtime data. On the other hand, KieSession is a ‘space’ where runtime data are stored and operations are executed. While creating a KieBase can be resource-intensive, the KieSession should be lightweight and created from the KieBase as needed.

Demo:

```
// Directly creating a KieSession from the KieContainer
KieSession kSession = kContainer.newKieSession();
```

Detailed kmodule.xml Configuration

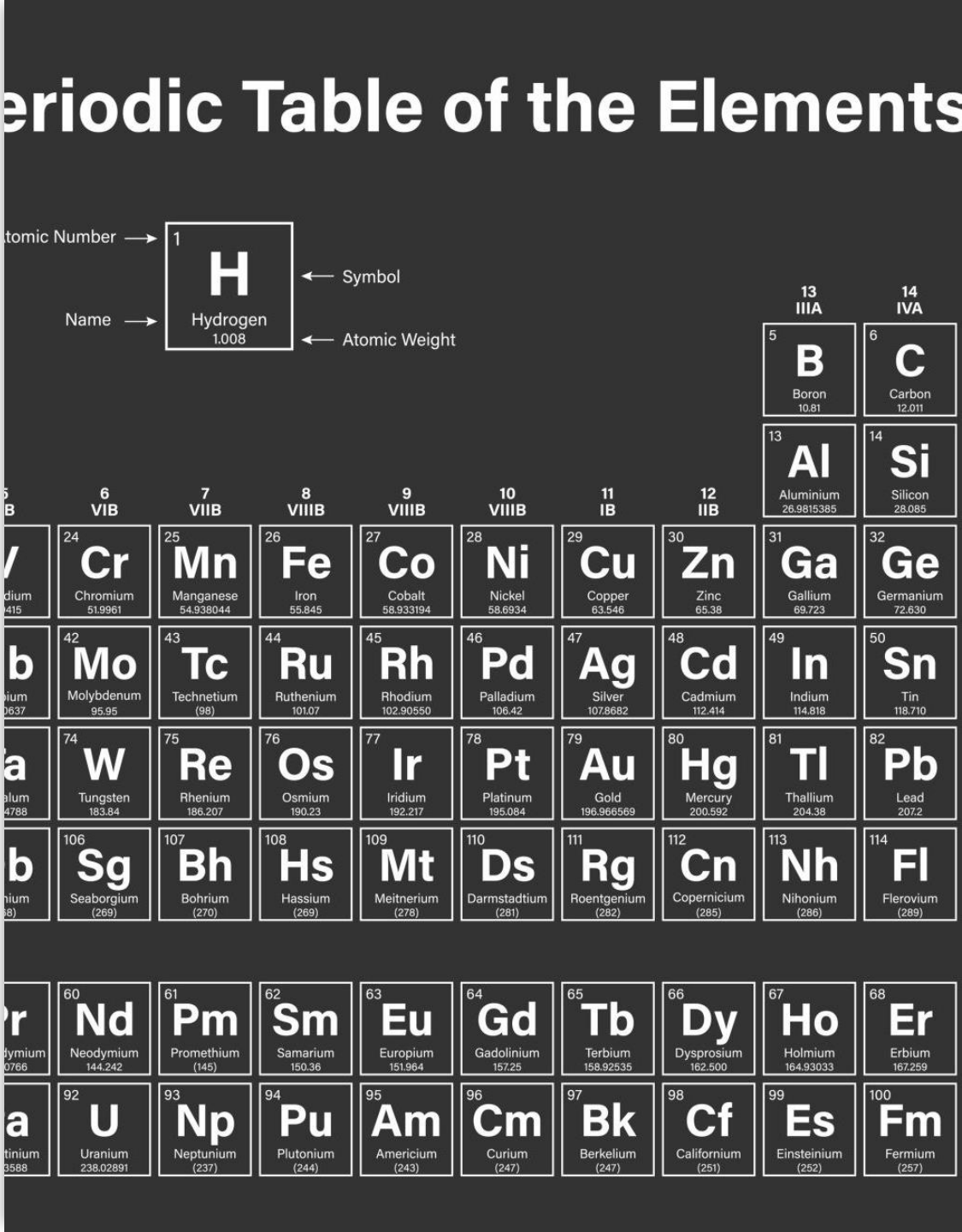
Tailoring KieBase and KieSession

The kmodule.xml file allows for in-depth customization of KieBases and the KieSessions they create. By defining properties such as event processing modes, default behaviors, and declarative agendas, users can adjust the KieBase to fit specific project needs. Further, each KieBase can be configured with multiple KieSessions, stateful or stateless, with various settings.

Detailed kmodule.xml Configuration

Demo:

```
<!-- A sample kmodule.xml file with advanced configuration -->
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true"
eventProcessingMode="cloud">
    <ksession name="KSession2_1" type="stateful" default="true"/>
  </kbase>
  <kbase name="KBase2" includes="KBase1">
    <ksession name="KSession3_1" type="stateful" default="false"
clockType="realtime"/>
  </kbase>
</kmodule>
```



Fetching KieSessions

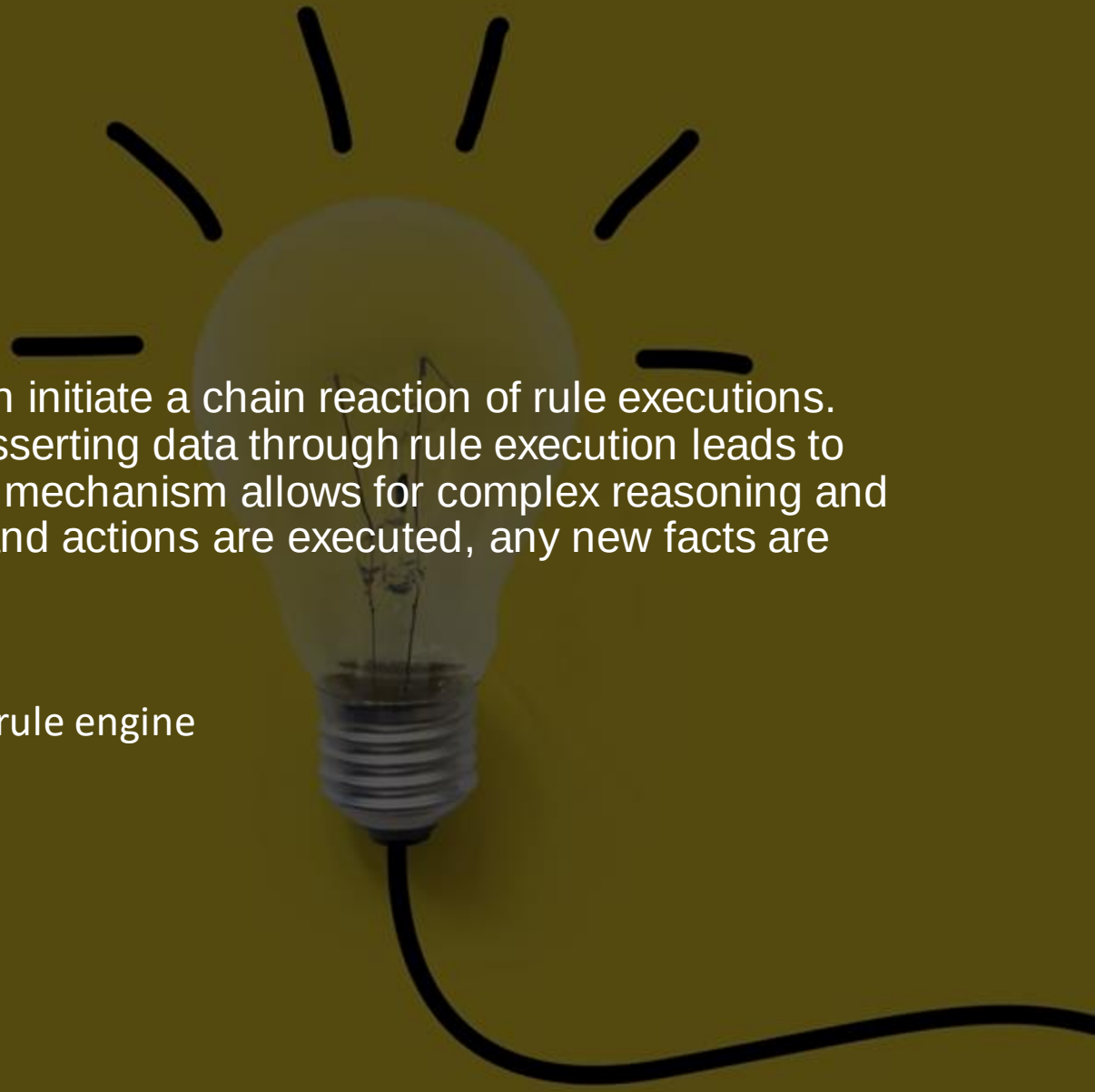
Instantiating Sessions for Rule Execution

Retrieving KieSessions involves interacting with the KieContainer and specifying the configurations set in the kmodule.xml. KieSessions, when fetched and populated with data, are where the ‘magic’ happens: rules are fired, and decisions are made based on the input facts.

Demo:

```
KieSession kSession2_1 = kContainer.newKieSession("KSession2_1");
KieSession kSession3_1 = kContainer.newKieSession("KSession3_1");
```

Inference



Rule-Driven Inference

In rule-based systems, the introduction of new data can initiate a chain reaction of rule executions. Drools handles this by performing inference—where asserting data through rule execution leads to new knowledge that influences subsequent rules. This mechanism allows for complex reasoning and decision-making, ensuring that as rules are triggered and actions are executed, any new facts are integrated into the decision-making process.

Demo:

```
// Example of a rule that infers new data within the Drools rule engine
rule "Speeding Infraction"
when
    $driver: Driver()
    $violation: Violation(speed > 100, driver == $driver)
then
    insertLogical(new SpeedingInference($driver));
end
```

Truth Maintenance

Ensuring Consistency in Rule Applications

Truth maintenance in Drools justifies each data assertion, maintaining consistency and truthfulness across rule executions. When a rule contradicts previous ones, Drools resolves the contradiction by relying on previously calculated conclusions. This helps identify inconsistencies and manage facts inserted either statedly or logically, ensuring the integrity of the engine's knowledge base.

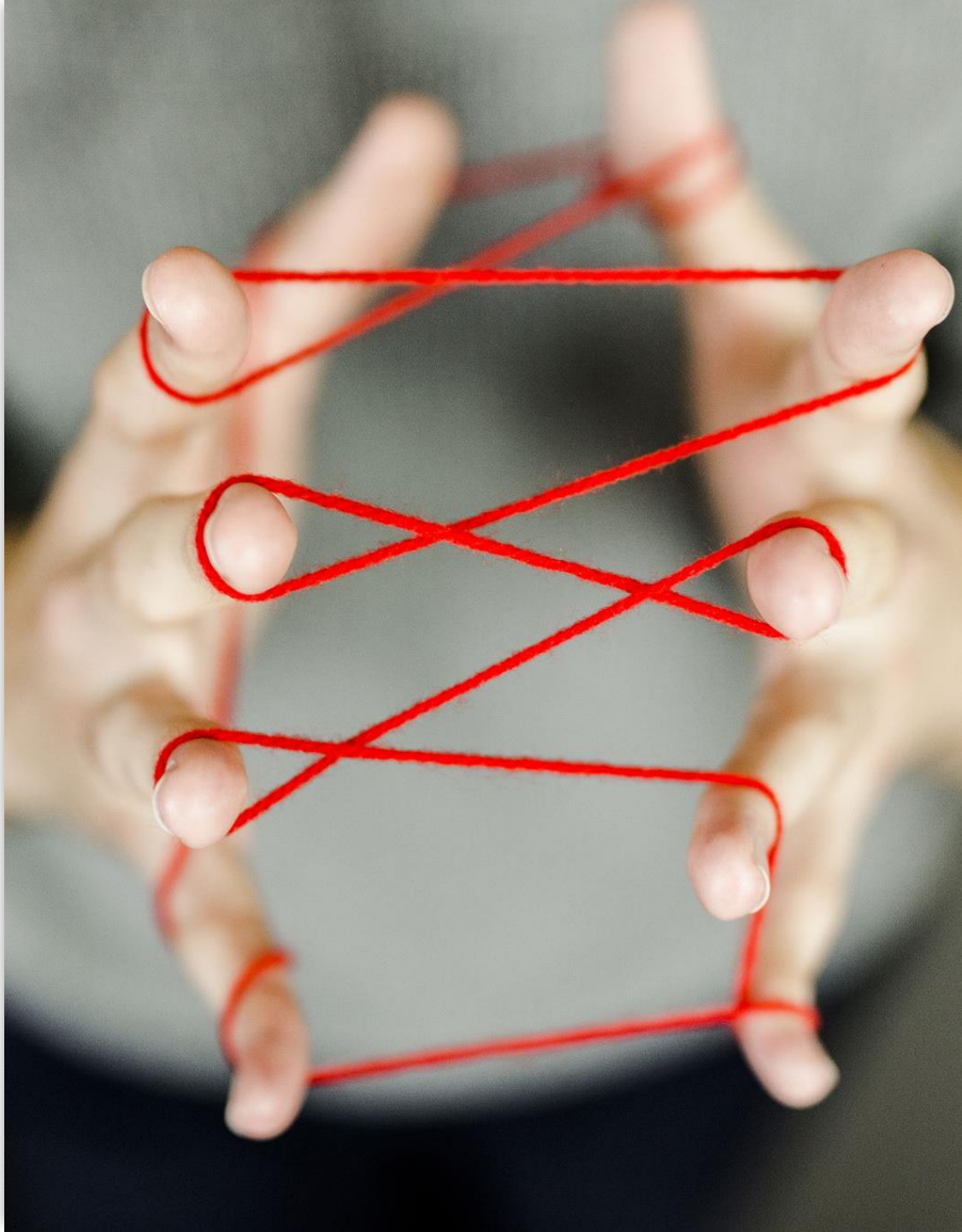
Demo:

```
// Example of using truth maintenance with stated insertions in Drools
rule "Increase Insurance Premium"
when
    $driver: Driver(accidents > 2)
then
    insert(new InsuranceIncrease($driver));
end
```

Logical vs. Stated Insertions

Stated vs. Logical Fact Insertions

Drools makes a distinction between stated insertions and logical insertions. Stated insertions, created with 'insert()', typically require explicit retraction when they're no longer needed. On the other hand, logical insertions, made using 'insertLogical()', are maintained by the engine as long as their justifying conditions are true, which enables automatic cleanup of facts when the conditions are no longer met.



Demo:

```
// Using logical fact insertions in a Drools rule
rule "Qualified for Discount"
when
    $customer: Customer(loyaltyPoints > 1000)
then
    insertLogical(new
DiscountQualification($customer));
end
```

With this system, we can build rules that not only execute based on the current state of facts but also adapt as new information is inferred or retracted, keeping the system dynamically in line with the truth of the domain it represents

Logical vs. Stated Insertions

Advanced Attributes to Steer Rule Firing

Drools provides advanced rule attributes that help manage how and when rules are executed. Attributes such as 'salience' can dictate the firing order of rules, 'no-loop' can prevent rules from re-activating themselves, and 'agenda-group' can group rules for focused execution. Understanding these attributes is fundamental for sophisticated control over rule execution.

Leveraging Rule Attributes

Leveraging Rule Attributes

Demo:

```
// Using salience to define rule execution order
rule "High Priority Rule" salience 100
when
   // conditions
then
   // actions
end

rule "Default Priority Rule"
when
   // conditions
then
   // actions
end
```

Agenda Groups and Activation Groups

Grouping for Organized Execution

Agenda groups allow you to categorize rules into different sets that can be focused on one at a time, while activation groups limit rule execution to only one rule within the group being activated. These groupings help prevent rule conflicts and manage complex decision flows by controlling rule execution based on the data context.

Agenda Groups and Activation Groups

Demo:

```
// Dividing rules into agenda groups
rule "Rule in Agenda Group A" agenda-group "Group A"
when
   // conditions
then
   // actions
end

// Defining activation groups to restrict rule firing
rule "Only one Activation" activation-group "Exclusive Group"
when
   // conditions
then
   // actions
end
```

Temporal Reasoning with Drools Fusion

Time-Aware Rule Execution

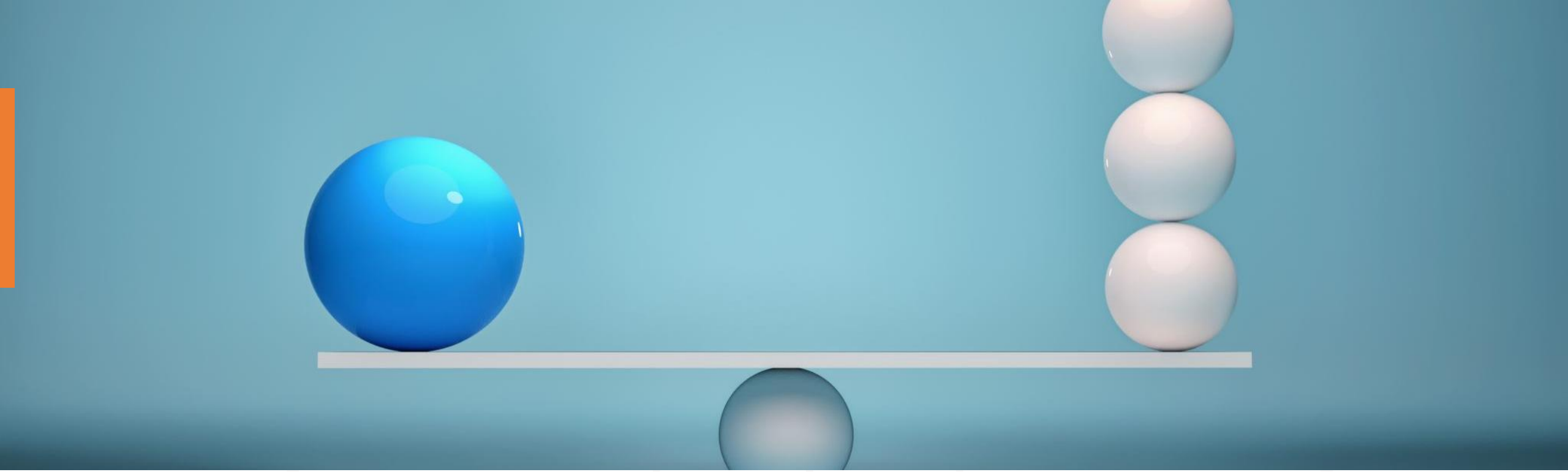
Complex event processing (CEP) and temporal reasoning are achieved in Drools through its extension, Drools Fusion. By leveraging temporal operators and specifying event processing modes, rules can react to the temporal relationships between events, allowing for scenarios like detecting patterns over time, managing windows of time, and performing event correlation.

Temporal Reasoning with Drools Fusion

Demo:

```
// Example of a temporal reasoning rule in Drools Fusion
declare StockTick
   @role(event)
end

rule "Detect Rapid Stock Movement"
when
   $tick1 : StockTick( symbol == "ACME", price > 100 )
   $tick2 : StockTick( symbol == "ACME", this after[5m,10m] $tick1, price < 90 )
then
   // actions for rapid price drop
End
```

These advanced rule execution controls allow for granular management of rule flow and decision-making within Drools, enabling the development of responsive and intelligent rule-based systems.



Fact Equality in Drools

Identity vs. Equality

In Drools, the way facts are compared and identified plays a crucial role in how rules are triggered. Two primary modes manage fact handling: 'identity' and 'equality'. In identity mode, each object is considered a separate fact, even if their content is the same. Equality mode allows for object content comparison, treating objects with the same values as equal, which can affect rule firing based on existing facts in the session.

Demo:

```
// Configuring equality behavior in kmodule.xml
<kbase name="equalityKBase" equalsBehavior="equality">
   <!-- KieBase configuration -->
</kbase>
```

Propagation Modes

Lazy, Immediate, and Eager Fact Handling

Drools supports different fact propagation modes that influence the order and timing of rule execution: 'Lazy' batches facts and processes them effectively, 'Immediate' processes facts as soon as they're inserted, and 'Eager' also batches facts but before rule execution. These modes are selected based on the rule attributes and can significantly influence performance and rule outcomes.

Propagation Modes

Demo:

```
// Example demonstrating different propagation modes

// Lazy Fact Propagation
rule "Lazy Propagation Rule"
when
   // conditions
then
   // actions using lazy propagation
end

// Immediate Fact Propagation
rule "Immediate Propagation Rule" salience 100
when
   // conditions
then
   // actions executed immediately after fact insertion
end
```

Utilizing Propagation Modes

Choosing the Right Propagation Mode

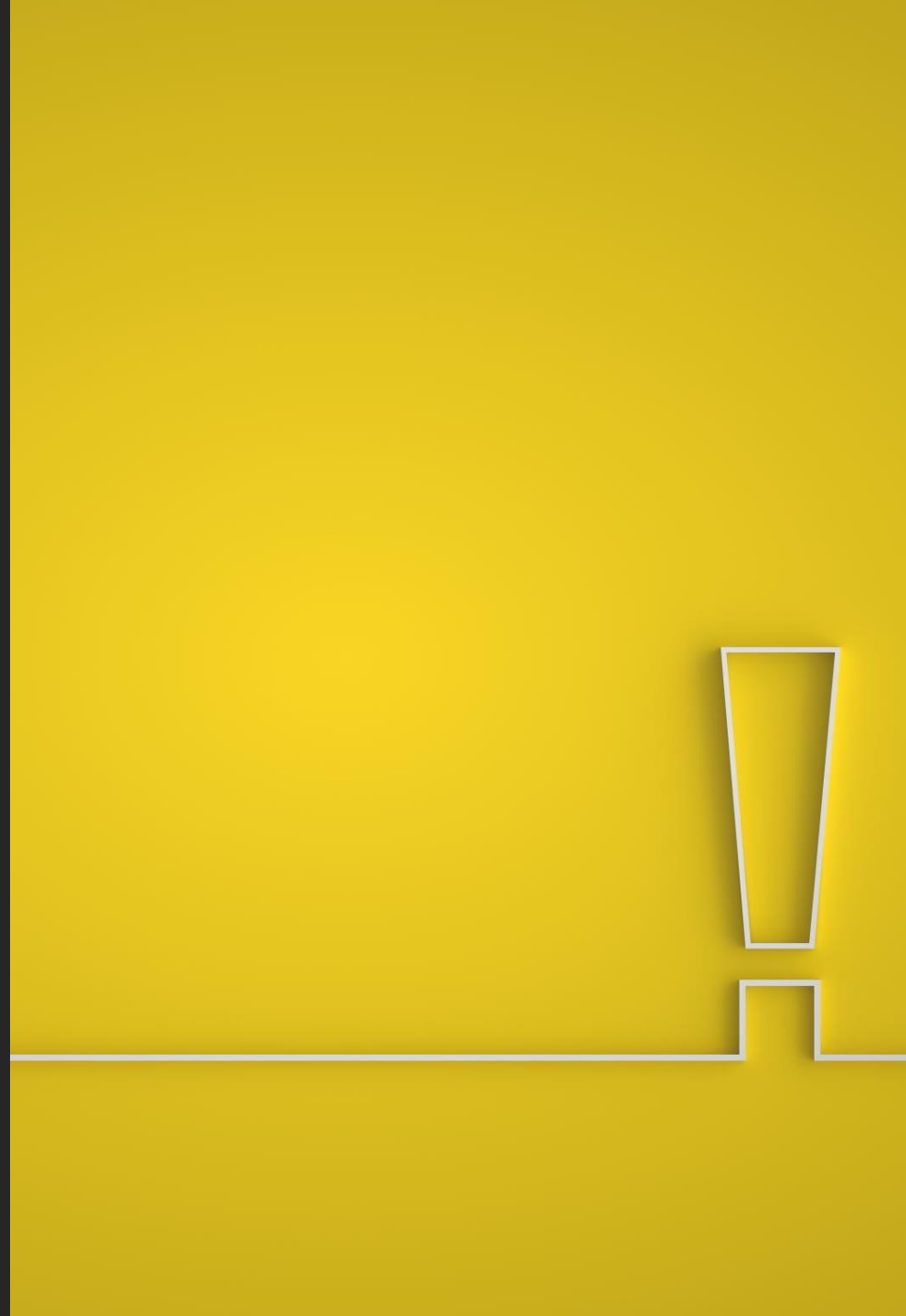
Selecting the appropriate fact propagation mode is paramount for achieving desired rule execution behavior. Understanding the context of your application's needs will dictate whether you should opt for the laziness of batch processing or the immediacy of reacting to data as it comes in. Selecting the right mode ensures rules are fired in an order and occurrence that reflect the logical and temporal needs of your business logic.

Utilizing Propagation Modes

Demo:

```
// Example of specifying propagation with the no-loop attribute

rule "No Loop Propagation"
no-loop
when
   $fact: FactType()
   // additional conditions
then
   // perform actions without concern for looping caused by fact modifications
end
```

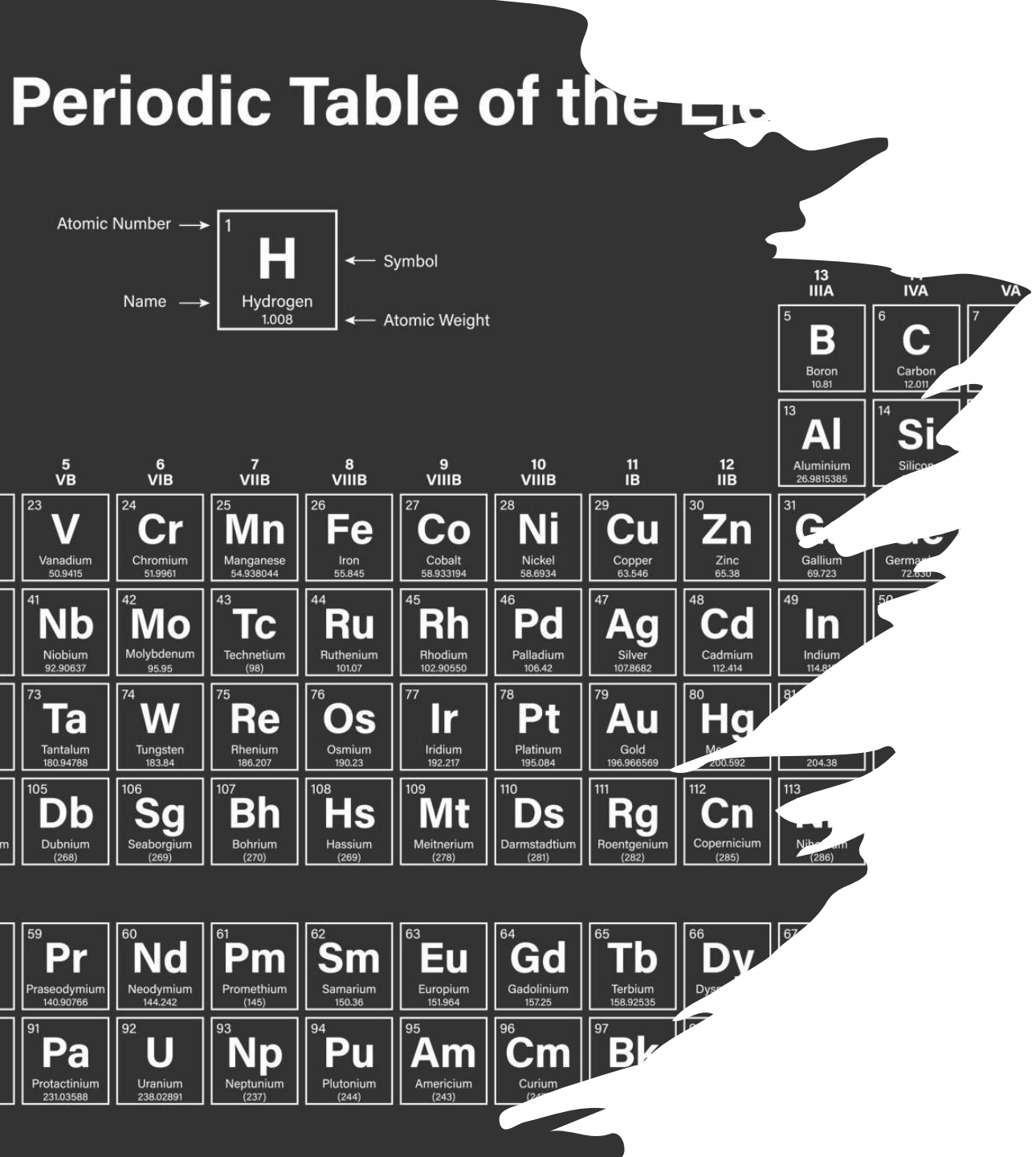
These slides delve into the advanced aspects of Drools regarding how facts are managed and compared, as well as the various modes available for propagating facts throughout the rule engine. Mastery of these concepts is key for optimizing the performance and predictability of rule-based applications.



Complex Event Processing (CEP)

Introducing CEP

Complex Event Processing (CEP) is crucial in scenarios where events have strong temporal relationships and patterns need prioritization and analysis over individual events. Drools offers robust CEP capabilities, allowing for event processing with proper semantics, detection, correlation, aggregation, and composition. Additionally, CEP in Drools supports event stream processing and can handle temporal constraints, sliding windows of significant events, and large volumes of event data.



Complex Event Processing (CEP)

Demo:

```
// A simple CEP rule in Drools
declare StockTrade
   @role(event)
end
rule "Detect High Trade Volume"
when
   Number( intValue > 10000 ) from accumulate(
      $trade: StockTrade( company == "ACME", volume > 1000 ) over window:time( 1h ),
      count($trade)
   )
then
   // actions to be taken when high trade volume detected
end
```

CEP Use Cases and Features

Use Cases and Optimizations for CEP

With CEP, Drools can be applied to various use cases, such as real-time stock trading analysis or environmental monitoring within a server room. Matching patterns and non-functional requirements like response times are key to these scenarios. Drools provides the necessary infrastructure for seamless integration with enterprise applications and supports the creation and management of rules to detect complex patterns of events efficiently.

CEP Use Cases and Features

Demo:

```
// CEP rule for environmental monitoring in Drools
declare TemperatureReading
   @role(event)
end

rule "Monitor Server Room Temperature"
when
   $event: TemperatureReading( serverRoom == "SR1", temperature > 30 ) over window:time( 30m )
then
   // action to be performed when temperature threshold exceeded
end
```

Setting Up CEP with Drools

Tuning Drools for Time-Sensitive Logic

Configuring Drools for CEP involves recognizing events and defining rules that leverage temporal constraints. This setup enables the Drools engine to process events continuously and allows developers to define windows for detecting event patterns within a specified timeframe, ensuring that rules react to events as they unfold over time.

Setting Up CEP with Drools

Demo:

```
// Demonstrating window and temporal constraints in a Drools CEP rule
declare Position
    @role(event)
end
rule "Detect Movement Within Area"
when
    $enter: Position( area == "RestrictedZone" ) over window:time( 10s )
    $leave: Position( area != "RestrictedZone", this after[0s, 10s] $enter )
then
    // Respond to the event that an object entered and stayed in a restricted area
end
```

Identifying Performance Bottlenecks

Diagnosing Slowdowns

The first step in tuning Drools performance is identifying the bottlenecks that cause slowdowns during rule execution. This can include the number of rules, complexity of rule conditions, or the size and shape of the fact model. Tools like profilers and Drools' built-in metrics can highlight the slowest parts of the rule execution process, guiding targeted optimizations to improve overall performance.

Optimization and Best Practices

Strategies for Speed and Scalability

Several best practices can enhance Drools performance, including simplifying rule conditions, using efficient data structures, employing indexing in rules, and avoiding global variables. Beyond these, carefully structuring rules to minimize unnecessary evaluations and using agenda groups to control rule execution order are effective strategies. Always profile before and after making changes to validate performance gains.

The Phreak Rule Algorithm

Revolutionizing Rule Evaluation

Drools adopts the Phreak (PHRete and lEAK compaction) algorithm, a highly efficient rule network algorithm that supersedes the classic Rete algorithm for most scenarios. Phreak enhances performance by lazily evaluating rule clauses, significantly reducing memory footprint during rule execution. Understanding and making use of the Phreak algorithm allows developers to write rules that execute more efficiently, particularly in scenarios with large sets of data and complex rules.

Implementing and Testing CEP Rules

Making Rules React to Events

With the event model in place, construct CEP rules that react to the event stream. These rules will use temporal operators to express constraints and relations between events, such as "every", "not", and "after". After rules are implemented, create a simulated stream of events to validate the functionality of your application, verify that rules trigger as expected, and adjust the logic as necessary.

Implementing and Testing CEP Rules

Demo:

```
// Sample CEP rule that correlates different events in Drools
rule "High Value Transaction Alert"
when
    $event1 : TransactionEvent(type == "Credit", amount > 1000) over
window:length(10)
    $event2 : TransactionEvent(this after[0s, 30s] $event1, type ==
"Withdrawal", amount > 5000)
then
    // respond to event pattern indicating potentially fraudulent activity
end
```

Practical Exercise

Real-Time Event Processing with Drools

Now it's time to put your knowledge into practice by building a CEP application utilizing Drools. You'll be simulating a real-time monitoring system that processes and correlates events as they happen. This exercise will walk you through setting up a Drools session configured for event processing, defining events, and writing rules that react to patterns of these events using the advanced features of Drools Fusion.

See Lab 04 -

DROOLS 8

MODULE 05:
DROOLS RULE LANGUAGE (DRL)

MASTERING THE LANGUAGE OF DROOLS

Presented by John Paul Franke

Course Overview

In this module, we delve into the Drools Rule Language (DRL), an essential component of the Drools rule engine. We will explore the syntax, structure, and capabilities of DRL, providing a comprehensive understanding of how to craft effective rules. The module will guide you through the nuances of rule creation, from basic syntax to advanced features, ensuring a solid grasp of DRL's potential in automating complex decision-making processes. By the end of this overview, you will be well-equipped to write, manage, and optimize rules in Drools, laying a foundation for more advanced topics in rule management and execution.

Learning Objectives

This Module's Goals

- Gain a comprehensive understanding of Drools Rule Language (DRL) syntax and structure.
- Learn how to create effective DRL rules for various scenarios.
- Explore advanced features of DRL to enhance rule capabilities.
- Understand how to use rule units and queries within DRL.
- Master error handling and troubleshooting techniques in DRL.
- Discover best practices for optimizing DRL performance.
- Prepare for practical exercises focused on building comprehensive rule sets in Drools.

Learning Objectives

This Module's Goals

- Understand the steps involved in building a Drools project, including rule authoring and testing.
- Learn how to deploy Drools applications across different environments.
- Explore how to run Drools applications and manage the execution of rules.
- Discover the use of KieScanner for dynamic updates in rule management.
- Dive into the concepts of persistence and transactions within Drools.
- Apply the learned concepts by building and deploying a sample Drools application.
- Prepare for the hands-on exercises that will reinforce your Drools knowledge.
- Get ready to showcase your understanding of the module's content through the assessment.

Learning Objectives

This Module's Goals

- Understand the steps involved in building a Drools project, including rule authoring and testing.
- Learn how to deploy Drools applications across different environments.
- Explore how to run Drools applications and manage the execution of rules.
- Discover the use of KieScanner for dynamic updates in rule management.
- Dive into the concepts of persistence and transactions within Drools.
- Apply the learned concepts by building and deploying a sample Drools application.
- Prepare for the hands-on exercises that will reinforce your Drools knowledge.
- Get ready to showcase your understanding of the module's content through the assessment.

Introduction to DRL

What is Drools Rule Language (DRL)?

Drools Rule Language (DRL) is the foundation of the Drools Rule Engine. It's a domain-specific language tailored to express business rules in a format that is both readable and writable by business analysts and developers. DRL allows the definition of complex logic in a way that mimics natural language, making it more accessible and easier to maintain.

Key Points:

- **Syntax and Semantics**: DRL uses a syntax resembling natural language, facilitating easier understanding and modification of rules.
- **Fact Model**: DRL operates on a fact model, where facts represent business data and are the primary elements that trigger rule execution.
- **Rule Structure**: A typical DRL rule consists of a `when` part (conditions) and a `then` part (actions).

DRL Syntax

Breaking Down DRL Syntax

Each DRL rule comprises distinct components, each playing a vital role in the rule's execution.

- **Rule Name**: Uniquely identifies the rule.
- **Attributes**: (Optional) Control rule execution behavior (e.g., salience for priority).
- **When Clause**: Defines conditions using facts and logical expressions.
- **Then Clause**: Specifies actions to execute when the when conditions are met.
- **End Keyword**: Marks the end of the rule.

DRL Syntax

Breaking Down DRL Syntax

Example:

```
rule "High-Value Transaction"
    when
        $transaction : Transaction(amount > 1000)
    then
        System.out.println("High-value transaction detected
: " + $transaction);
end
```

DRL Syntax

Breaking Down DRL Syntax

Example 01:

```
rule "High-Value Transaction"
    when
        $transaction : Transaction(amount > 1000)
    then
        System.out.println("High-value transaction detected
: " + $transaction);
end
```

DRL Syntax

Breaking Down DRL Syntax

Example 02:

```
rule "Eligible for Discount"
    when
        $customer : Customer(age > 60)
    then
        applyDiscount($customer);
end
```

This rule applies a discount when the customer's age is over 60.

Advanced DRL Syntax

Advanced features of DRL include:

1. **Complex Pattern Matching**: Using logical operators (and, or, not) for intricate conditionals.
2. **Accumulate Functions**: Aggregating data, like summing values or counting occurrences.
3. **Temporal Operators**: Handling events based on time constraints.
4. **Rule Flows**: Defining the order and dependencies between different rules.

Advanced DRL Syntax

Advanced features of DRL include:

- **Complex Pattern Matching**: Using logical operators (and, or, not) for intricate conditionals.
- **Accumulate Functions**: Aggregating data, like summing values or counting occurrences.
- **Temporal Operators**: Handling events based on time constraints.
- **Rule Flows**: Defining the order and dependencies between different rules.

Advanced DRL Syntax

Example:

```
rule "Frequent Buyer Reward"
    when
        $customer : Customer(loyaltyPoints > 500)
        $totalPurchases : Number() from accumulate(
            Purchase(customer == $customer), sum($purchase.amount)
        )
    then
        rewardCustomer($customer, $totalPurchases);
end
```

This rule rewards customers who have more than 500 loyalty points and a certain amount in purchases.

Structuring DRL Rules

Best Practices in Rule Structuring	
Effective structuring of DRL rules is essential for maintainability and performance:	**Modularization**: Group related rules for better organization and clarity.
	Naming Conventions: Use clear, descriptive names for rules and variables.
	Commenting: Document rules for better understanding and future reference.
	Efficient Conditions: Optimize conditions to prevent unnecessary evaluations.
Reusability: Create generic rules that can be applied to multiple scenarios.	

Structuring DRL Rules

Example:

```
// Rule for validating order quantity
rule "Validate Order Quantity"
    when
        $order : Order(quantity <= 0)
    then
        rejectOrder($order);
end
```

This rule ensures that order quantities are valid and rejects any order with a non-positive quantity.

DRL Rule Optimization Techniques

Techniques for Efficient Rule Execution	**Simplify Conditions**: Break down complex conditions into simpler, more efficient ones.
	Using Indexing and Hashing: Apply indexing on frequently evaluated fields in rule conditions.
	Avoiding Redundant Evaluations: Use no-loop, lock-on-active to prevent unnecessary re-evaluation of rules.
	Optimizing Fact Handling: Minimize the number of facts in working memory and efficiently manage fact insertions and updates.

DRL Rule Optimization Techniques

Example:

```
// Optimized rule with simplified conditions
rule "Efficient Stock Check"
    when
        $product : Product(stock < threshold, stoc
k > 0)
    then
        reorderProduct($product);
end
```

This rule efficiently triggers product reordering based on optimized stock conditions.

Best Practices in DRL Rule Development

Best Practices for Developing DRL Rules

Adhering to best practices in DRL rule development enhances reliability:

1. **Testing and Validation**: Rigorously test rules to ensure they work as expected in various scenarios.
2. **Version Control**: Use version control systems to manage changes in rules, allowing easy rollback and tracking of modifications.
3. **Continuous Integration**: Integrate rules within a CI/CD pipeline for continuous testing and deployment.
4. **Collaboration**: Encourage collaboration between developers and business analysts to ensure that rules accurately reflect business logic.

Advanced Rule Attributes in DRL

Utilizing Advanced Rule Attributes

Advanced attributes in DRL provide more control and functionality to rules:

1. **Salience**: Determines the firing order of rules, with higher values having priority.
2. **Agenda-group**: Groups rules into categories, allowing selective execution.
3. **Activation-group**: Ensures that only one rule from a group is executed.
4. **Timers**: Triggers rules based on time events or schedules.
5. **Auto-focus**: Automatically focuses an agenda group when a rule in that group becomes active.

Advanced Rule Attributes

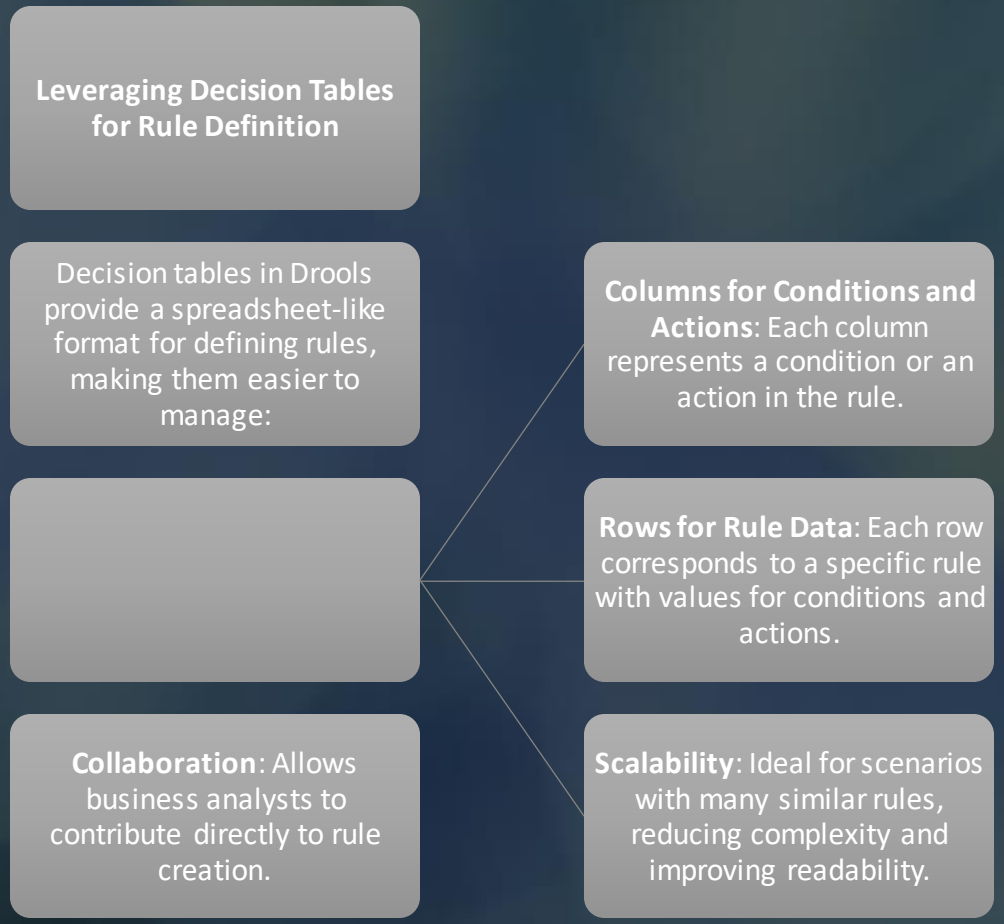
Utilizing Advanced Rule Attributes

Example:

```
// Using advanced attributes
rule "Priority Customer Check" salience 10 agenda-group
"CustomerStatus"
    when
        $customer : Customer(status == "Gold")
    then
        // actions for priority customers
end
```

This rule has a higher priority and belongs to the "CustomerStatus" agenda group.

Decision Tables in DRL



Decision Tables in DRL

Leveraging Decision Tables for Rule Definition

Example:

```
// Represented as a spreadsheet

| Rule       | Condition              | Action                    |
|------------|------------------------|---------------------------|
| Rule 1     | Customer Type: VIP     | Apply Discount: 20%       |
| Rule 2     | Customer Type: New     | Apply Discount: 10%       |
```

This decision table applies different discounts based on customer type.

Event Processing in Drools

Real-Time Decision Making

Event processing in Drools allows for the handling of complex, real-time scenarios:

- **Stream Mode**: Processes facts as events, enabling real-time decision making.
- **Temporal Operators**: Supports operators like "after", "before", "during", and "overlaps" for time-based conditions.
- **Complex Event Processing (CEP)**: Allows for pattern recognition over time, suitable for monitoring and alerting scenarios.

Event Processing in Drools

Example:

```
// Temporal rule example
rule "High Temperature Alert"
    when
        $event : TemperatureEvent(temperature > 100) ov
er window:time(1m)
    then
        alert("High temperature detected!");
end
```

This rule generates an alert if high temperatures are detected over a one-minute window.

Introduction to Rule Units in DRL

Modularizing Rules for Better Organization	
Rule Units in Drools are a method of modularizing rules into cohesive units:	**Definition**: A Rule Unit encapsulates a set of rules and the data they operate on.
	Purpose: Organizes rules into logical groups, improving maintainability and readability.
	Data-Driven Execution: Rule Units can be executed with specific data sets, allowing more controlled and targeted rule execution.

Introduction to Rule Units in DRL

Modularizing Rules for Better Organization
Example:

```
// Example of a Rule Unit declaration
unit MyRuleUnit;
rule "Eligible for Offer" ruleflow-group "Offers"
    when
        $customer : Customer(age > 25)
    then
        applySpecialOffer($customer);
end
```

This Rule Unit targets customers over 25 for special offers.

Utilizing Queries in DRL

Extracting Information with DRL Queries

Queries in DRL provide a way to retrieve information from the working memory:

Functionality: Similar to database queries, they allow extraction of data based on specific conditions.

Use Cases: Useful for reporting, diagnostics, and checking the state of the working memory.

Syntax: Defined with the `query` keyword, followed by a name and optional parameters.

Example:

```
// Query example in DRL
query "Find High Risk Customers"
    $customer : Customer(riskRating > 8)
end
```

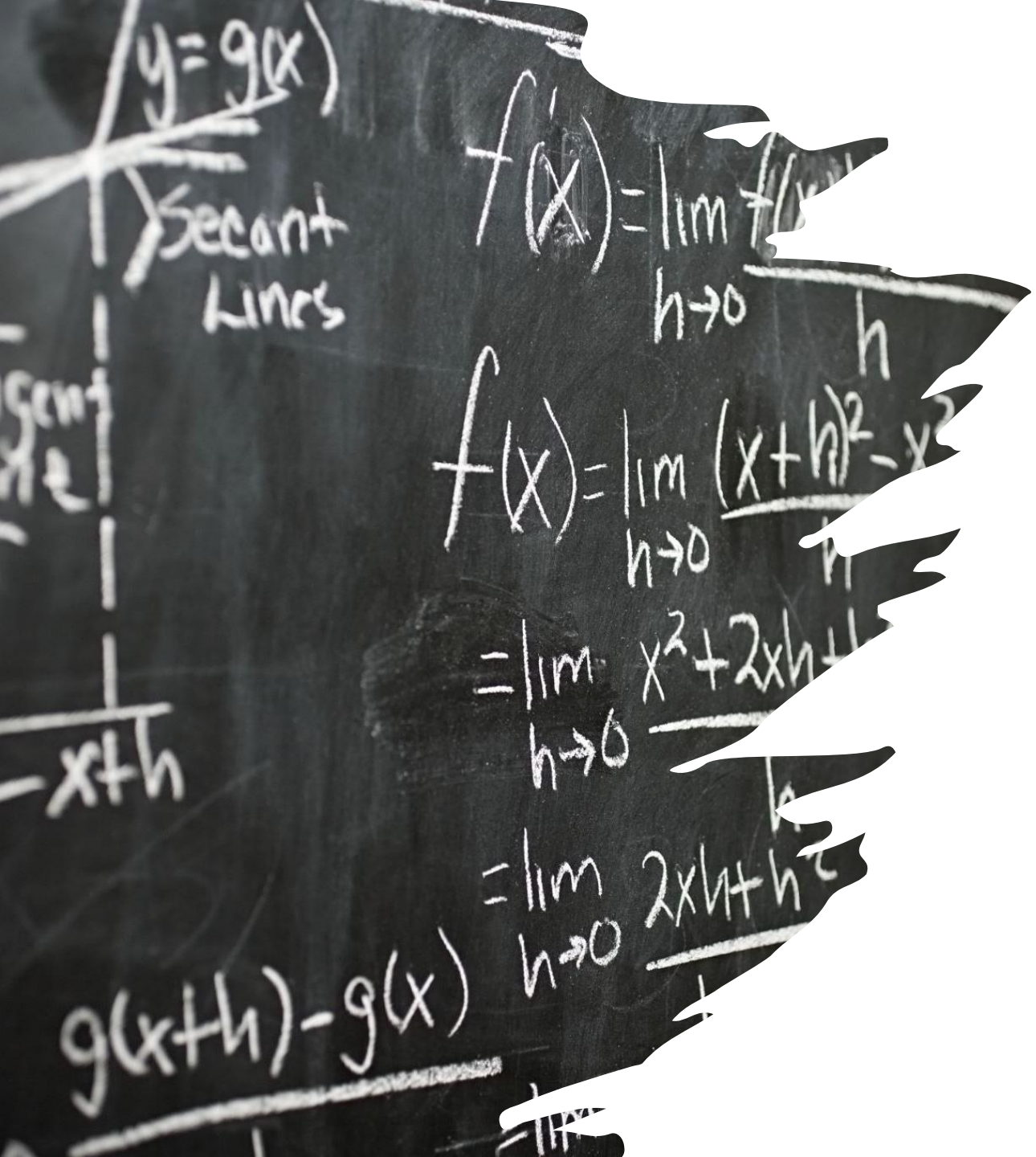
This query retrieves customers with a risk rating higher than 8.

Rule Units and Queries – Best Practices

Enhancing DRL Effectiveness

To maximize the effectiveness of Rule Units and Queries:

1. **Cohesion**: Keep Rule Units cohesive with rules closely related in logic and purpose.
2. **Isolation**: Isolate different aspects of business logic in separate Rule Units for clarity.
3. **Optimization**: Use queries for data extraction without side effects, keeping them efficient and focused.
4. **Integration**: Integrate Rule Units and queries into larger business processes for streamlined decision-making.



Rule Units and Queries – Best Practices

Example:

```
// Integrating a query in a Rule Unit
unit CustomerRiskAssessment;
query "HighValueCustomers"
    $customer : Customer(value > 100000)
end

rule "Assess High Value Customer" rulefl
ow-group "Assessment"
    when
        $customer : Customer() from High
ValueCustomers()
    then
        assessRisk($customer);
end
```

This demonstrates the use of a query within a Rule Un it to assess the risk of high-value customers

Troubleshooting Tips for DRL

Resolving Common Rule Issues

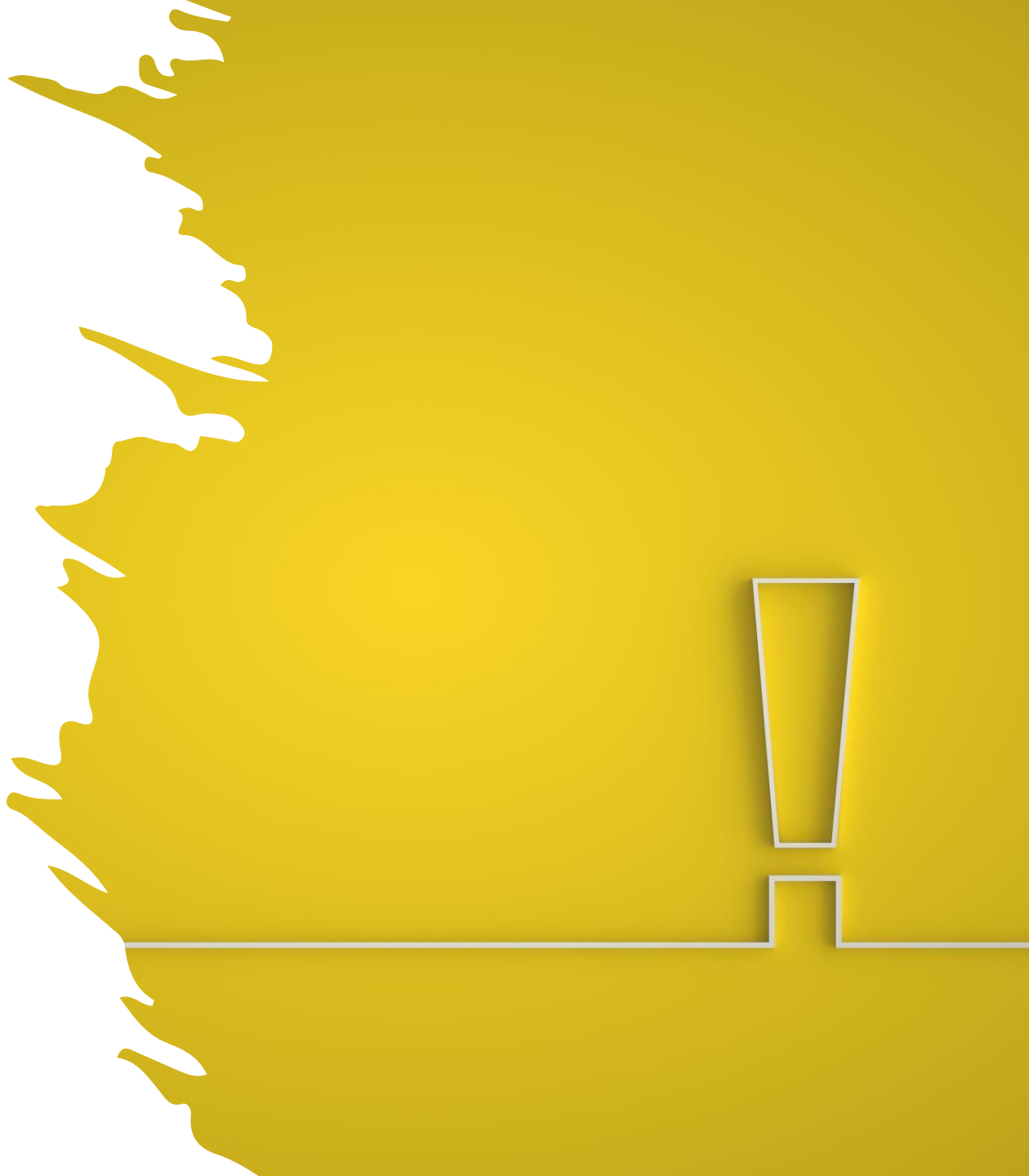
Common issues in DRL and their troubleshooting tips:

1. **Rule Not Firing**: Verify the conditions and the data in the working memory. Ensure that all facts are inserted correctly.

2. **Performance Bottlenecks**: Profile rule execution to identify slow rules. Optimize rule conditions and consider indexing.

3. **Unexpected Behavior**: Use logging to trace rule execution and understand which rules are firing and why.

4. **Syntax Errors**: Check for typos and incorrect syntax, especially in complex expressions.



Troubleshooting Tips for DRL

Example:

```
// Example of adding logging for troubleshooting
rule "Eligibility Check"
when
$customer : Customer()
then
log("Checking eligibility for customer: " + $customer);
// additional actions
end
```

This rule logs the process of checking a customer's eligibility, aiding in troubleshooting.

Debugging Best Practices in DRL

Efficiently Identifying and Resolving Rule Issues

To efficiently debug rules in DRL, follow these best practices:

1. **Incremental Development**: Build and test rules incrementally to identify issues early.
2. **Use of Debugging Tools**: Leverage Drools debugging tools for step-by-step execution analysis.
3. **Comprehensive Testing**: Create comprehensive test cases that cover various scenarios and edge cases.
4. **Peer Review**: Have rules reviewed by peers for a fresh perspective on potential issues.

Debugging Best Practices in DRL

Example:

```
// Implementing a testing appr
oach for rules
rule "Discount Application"
    when
        $order : Order(total >
500)
    then
        applyDiscount($order,
10);
        log("Discount applied
to order: " + $order);
end
```

This rule includes comprehensive logging to assist in debugging the discount application logic.

Identifying Performance Issues in DRL

Key Strategies for Efficient Rule Execution

Effective performance optimization in DRL starts with identifying potential bottlenecks:

1. **Profiling Rule Execution**: Use profiling tools to measure rule execution time and identify slow rules.
2. **Analyzing Data Patterns**: Examine the patterns and volume of the data processed by the rules.
3. **Evaluating Rule Complexity**: Check for overly complex or deeply nested conditions.
4. **Monitoring Memory Usage**: Assess the memory footprint during rule execution, especially for large data sets.

Identifying Performance Issues in DRL

Example:

```
// Using logging to measure rule
execution time
rule "Order Processing Time"
    when
        $order : Order()
    then
        long startTime = System.c
urrentTimeMillis();
        processOrder($order);
        long endTime = System.cur
rentTimeMillis();
        log("Order processed in "
+ (endTime - startTime) + " ms");
end
```

This rule logs the time taken to process an order, helping identify performance issues.

Enhancing DRL Rule Efficiency

Optimizing Rules for Better Performance

Implement these techniques to optimize DRL rules:

1. **Simplifying Conditions**: Break down complex conditions into simpler, more efficient ones.
2. **Using Indexing and Hashing**: Apply indexing on frequently evaluated fields in rule conditions.
3. **Avoiding Redundant Evaluations**: Use `no-loop`, `lock-on-active` to prevent unnecessary re-evaluation of rules.
4. **Optimizing Fact Handling**: Minimize the number of facts in working memory and efficiently manage fact insertions and updates.

Enhancing DRL Rule Efficiency

Example:

```
// Optimized rule with simplified conditions
rule "Efficient Stock Check"
    when
        $product : Product(stock < threshold, stock > 0)
    then
        reorderProduct($product);
end
```

This rule efficiently triggers product reordering based on optimized stock conditions.

Best Practices for DRL Performance

Ensuring Optimal Rule Engine Performance

Adopt these best practices to maintain high performance in your DRL implementation:

1. **Incremental Development and Testing**: Develop and test rules incrementally to catch performance issues early.
2. **Regular Refactoring**: Continuously refactor rules to adapt to changing requirements and maintain efficiency.
3. **Effective Memory Management**: Manage the working memory effectively to reduce memory leaks and overhead.
4. **Balancing Rule Complexity**: Balance between rule complexity and performance, keeping rules as straightforward as possible.

Best Practices for DRL Performance

Example:

```
// Regularly refactored rule for cla
rity and performance
rule "Customer Segmentation"
    when
        $customer : Customer(purchas
eHistory matches "Frequent")
    then
        segmentCustomer($customer, "
VIP");
end
```

This rule demonstrates a balance between clarity and performance in customer segmentation.

Practical Exercise Introduction

Hands-On DRL Development

This exercise will enhance your understanding of DRL by applying concepts learned throughout the module.

Exercise Overview:

1. **Scenario**: Develop a rule set for an e-commerce platform handling customer segmentation, product recommendations, and order processing.
2. **Objective**: Create efficient, clear, and effective rules that address various aspects of the e-commerce business logic.
3. **Tools**: Use the Drools Workbench for rule development and testing.

Starting Point:

- Define the fact model for customers, orders, and products.
- Outline the business scenarios to be addressed by the rules.

Exercise Part 1 - Customer Segmentation

Develop rules for segmenting customers into categories like 'New', 'Regular', and 'VIP' based on their purchase history and engagement.

Tasks:

1. **Define Conditions**: Identify attributes (e.g., purchase frequency, total spend) to segment customers.
2. **Write Rules**: Create DRL rules that classify customers into segments.
3. **Test Scenarios**: Validate the rules with different customer profiles to ensure accurate segmentation.

Exercise Part 1 - Customer Segmentation

Sample Rule:

```
rule "Identify VIP Customers"
    when
        $customer : Customer(totalSpend > 1000, orders > 10)
    then
        setCustomerSegment($customer, "VIP");
end
```

Exercise Part 2 – Product Recommendations and Order Processing

Enhancing Customer Experience and Efficiency

Create rules for personalized product recommendations and efficient order processing.

Product Recommendation Tasks:

1. **Analyze Purchase Patterns**: Determine patterns in customer purchases.
2. **Develop Recommendation Rules**: Write rules to suggest products based on customer preferences and purchase history.

Order Processing Tasks:

1. **Order Validation**: Ensure all orders meet certain criteria before processing.
2. **Dynamic Pricing**: Apply rules for discounts and special offers based on order size and customer segment.

Exercise Part 2 –
Product Recommendations and Order Processing

Sample Rule for Order Processing:

```
rule "Apply Bulk Order Discount"
    when
        $order : Order(totalItems > 20)
    then
        applyDiscount($order, 10); // 10% discount
for bulk orders
end
```

Conclusion

This concludes Module 05, "Drools Rule Language (DRL) Reference.” In this module, we’ve taken a deep dive into the syntax, structure, and advanced features of DRL, as well as strategies for creating effective and optimized rules. We've tackled practical exercises, enhancing our skills in rule writing and troubleshooting, and have explored best practices for performance optimization. Our foray into error handling and advanced rule features like rule units and queries has broadened our capabilities, preparing us to handle complex rule-based scenarios with proficiency. These skills will give us a solid foundation for utilizing DRL for various real-world challenges.

Summary

Key Points from Module 05 :

In This Module we…

- Learned the basics of Drools Rule Language (DRL), including how to write and structure rules.
- Explored more complex parts of DRL to manage difficult rule scenarios.
- Focused on making clear and efficient DRL rules.
- Looked at rule units and queries in DRL for more ways to use rules and find data.
- Learned how to fix common problems in DRL and keep rules working well.
- Discussed tips for making DRL rules work faster and better.
- Practiced what we learned by creating detailed rule sets.