***What is PL/SQL?***

PL/SQL (procedural language/structured query language) is a block-structured language developed by oracle that combines SQL with procedural programming constructs.

***Key features:***

1. Block structure: executes multiple queries in one block.

2. Procedural constructs: creates reusable program units like procedures, functions, and triggers.

3. Error handling: handles exceptions and errors in the program.

4. Reusable code: creates stored procedures, functions, and packages for repeated use.

5. Performance: reduces network traffic by executing multiple SQL statements in a single block.

***Differences between SQL and PL/SQL*:**

1. SQL is a single query language, while PL/SQL is a block-structured language.

2. SQL is declarative, while PL/SQL is procedural.

***PL/SQL block structure:***

1. Declare: optional section for declaring variables and constants.

2. Begin: mandatory section for executable statements.

3. Exception: optional section for handling exceptions.

***PL/SQL identifiers:***

1. Variables: declared with a name and data type.

2. Comments: single-line comments use "--", while multi-line comments use "/\* \*/".

***Practical Example:***

A PL/SQL block can take user input, perform calculations, and display output using DBMS\_OUTPUT.

*\*PL/SQL is a powerful tool for combining SQL with procedural programming capabilities, enabling developers to create sophisticated applications within the Oracle database.*

***What is a PL/SQL Block?***

A PL/SQL block is a unit of code that can be reused, containing business logic, conditional execution, and repetitive code.

***Syntax:***

A PL/SQL block consists of three parts:

1. DECLARE: Optional section for declaring variables, constants, and cursors.

2. BEGIN: Mandatory section for executable statements.

3. EXCEPTION: Optional section for handling exceptions.

***Types of Blocks:***

1. Anonymous Blocks: Unnamed blocks that can be executed only once.

2. Named Blocks: Reusable blocks with a name, such as procedures and functions.

***Example:***

A PL/SQL block can be used to retrieve data from a table, like a customer's details, and display the information.

PL/SQL blocks are useful for encapsulating logic, reducing code duplication, and improving performance.

**PL/SQL Basic Syntax:**

1. Block Structure: PL/SQL programs are divided into logical blocks of code, consisting of three sub-parts:

- Declarations: Optional section for declaring variables, cursors, and subprograms.

- Executable Commands: Mandatory section for executable PL/SQL statements.

- Exception Handling: Optional section for handling errors.

2. Identifiers: Constants, variables, exceptions, procedures, cursors, and reserved words.

3. Delimiters: Symbols with special meanings, such as +, -, \*, /, etc.

4. Comments: Explanatory statements that can be included in the code, ignored by the compiler.

5. Program Units: PL/SQL blocks, functions, packages, procedures, triggers, and types.

***Key Points:***

- PL/SQL blocks can be nested within other blocks.

- Every statement ends with a semicolon (;).

- Identifiers are not case-sensitive by default.

- Comments can be single-line (--) or multi-line (/\* \*/).

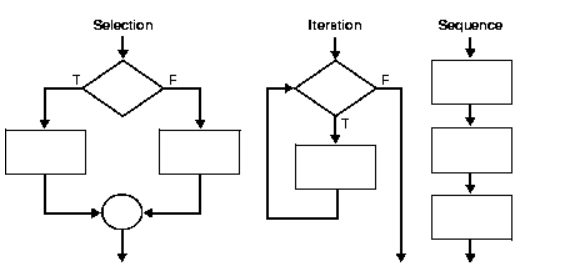
***Example:***

The "Hello World" example demonstrates a simple PL/SQL block that declares a variable, assigns a value, and prints the message using DBMS\_OUTPUT.

**PL/SQL Control Structures**

According to the *structure theorem*, any computer program can be written using the basic control structures shown in [Figure 4-1](https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/04_struc.htm#2608). They can be combined in any way necessary to deal with a given problem.

***Figure 4-1 Control Structures***



***Overview:***

PL/SQL control structures determine the flow of control through a program. They include:

1. Conditional Control: IF and CASE statements for decision-making.

2. Iterative Control: LOOP and EXIT statements for repetitive execution.

3. Sequential Control: GOTO and NULL statements for unconditional branching and no action.

***Key Points:***

- IF Statements: Used for conditional execution, with forms like IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

- CASE Statements: Used for selecting one sequence of statements based on a selector.

- LOOP Statements: Used for repetitive execution, with forms like LOOP, WHILE-LOOP, and FOR-LOOP.

- EXIT Statements: Used to complete a loop prematurely.

- GOTO Statements: Used for unconditional branching to a labeled statement.

- NULL Statements: Used to specify no action or to improve readability.

***Best Practices:***

- Use control structures to improve program readability and maintainability.

- Avoid overusing GOTO statements, as they can lead to complex, unstructured code.

- Use labels and dot notation to reference global variables or outer loop counters in nested loops.

**Error Handling**

PL/SQL error handling uses exceptions to catch and manage runtime errors, allowing programs to continue operating despite issues. Key aspects include:

- Predefined exceptions (e.g., ZERO\_DIVIDE, NO\_DATA\_FOUND)

- User-defined exceptions

- Exception handlers

- Reraising exceptions

\*You can also use procedures like RAISE\_APPLICATION\_ERROR to issue custom error messages.

**Cursors:**

PL/SQL cursors are pointers to a context area that holds rows returned by SQL statements. There are two types:

1. **Implicit Cursors**: Automatically created by Oracle for SQL statements without an explicit cursor.

2. **Explicit Cursors**: Programmer-defined for more control over the context area.

**Key aspects of explicit cursors include:**

- Declaring the cursor with a SELECT statement

- Opening the cursor to allocate memory

- Fetching rows from the cursor

- Closing the cursor to release memory

\*Cursors have attributes like %FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT to track their state and results.

**Procedures and Functions:**

PL/SQL procedures and functions are named PL/SQL blocks that can be executed multiple times. Key aspects include:

- Creating Procedures and Functions: Use CREATE PROCEDURE or CREATE FUNCTION statements to define standalone subprograms.

- Executing Procedures and Functions: Can be executed using anonymous blocks or CALL statements.

- Using Synonyms: Private and public synonyms can be created for procedures and functions to simplify access.

- Usage Notes: Considerations for replication, cache, object naming, and access rights.

\*Procedures and functions enable code reuse and modularity in PL/SQL programming.

**Packages:**

PL/SQL packages are powerful tools for organizing and managing code. Here are some key points to consider:

**What is a Package?**

A package is a schema object that groups logically related types, variables, constants, subprograms, cursors, and exceptions. It has a specification and a body, which can be compiled and stored in the database.

***Package Specification:***

The package specification declares public items that can be referenced from outside the package. It defines the interface to the package and is the only part of the package that is visible to other schema objects.

***Package Body:***

The package body defines the implementation of the public subprograms and cursors declared in the package specification. It can also declare private items that are not visible outside the package.

***Benefits of Packages:***

Packages provide several benefits, including:

- Modularity: Packages help to organize code into logical units, making it easier to maintain and modify.

- Easier Application Design: Packages can be designed and compiled independently of other schema objects, making it easier to develop and test applications.

- Hidden Implementation Details: Packages can hide implementation details, reducing dependencies between schema objects and making it easier to change or replace the implementation without affecting other parts of the application.

- Added Functionality: Packages can provide additional functionality, such as persistent variables and cursors, that can be shared across multiple subprograms.

- Better Performance: Packages can improve performance by reducing the overhead of compiling and loading code.

***Best Practices for Packages:***

Here are some best practices to keep in mind when working with packages:

- Keep Packages General: Design packages to be general and reusable, rather than specific to a particular application or use case.

- Design Specifications Before Bodies: Define the package specification before implementing the package body, to ensure that the interface is well-defined and stable.

- Declare Only Necessary Public Items: Only declare public items that need to be accessed from outside the package, to reduce dependencies and improve encapsulation.

- Use ACCESSIBLE BY Clause: Use the ACCESSIBLE BY clause to restrict access to the package to specific schema objects or users.

***Common Use Cases for Packages:***

Packages are commonly used in a variety of scenarios, including:

- API Development: Packages can be used to define APIs that provide a set of related functions and procedures that can be used by other schema objects or applications.

- Utility Functions: Packages can be used to provide utility functions that can be used across multiple schema objects or applications.

- Data Access: Packages can be used to encapsulate data access logic, providing a layer of abstraction between the application code and the underlying data structures.

\*Overall, packages are a powerful tool for organizing and managing code in PL/SQL, and can help to improve the maintainability, scalability, and performance of applications.

**Triggers:**

A trigger is a set of actions that are automatically executed in response to certain events, such as insert, update, or delete operations on a table. Triggers can be used to enforce complex business rules, audit changes, and prevent invalid data from being inserted into a table.

***Types of Triggers:***

- DML Triggers: Fired in response to insert, update, or delete operations on a table.

- System Triggers: Fired in response to system events, such as startup or shutdown.

- Compound Triggers: Can fire at multiple timing points, such as before and after a statement.

***Trigger Components:***

- Triggering Event: The event that causes the trigger to fire.

- Trigger Body: The code that is executed when the trigger fires.

- Timing Point: The point at which the trigger fires, such as before or after a statement.

***Benefits of Triggers:***

- Enforce Complex Business Rules: Triggers can enforce rules that cannot be defined using constraints.

- Audit Changes: Triggers can log changes to a table, providing a record of all modifications.

- Prevent Invalid Data: Triggers can prevent invalid data from being inserted into a table.

***Best Practices:***

- Use Triggers Judiciously: Triggers can impact performance, so use them only when necessary.

- Keep Triggers Simple: Complex triggers can be difficult to maintain and debug.

- Test Triggers Thoroughly: Triggers should be thoroughly tested to ensure they work as expected.

**Module 4 – Test driven development and Logging framework**

**What is Test Driven Development (TDD)?**

Test-Driven Development (TDD) Brief Overview

-Test-Driven Development is a software development method where you write automated tests before writing the actual code. It involves a repetitive cycle of writing tests, running tests, and refactoring code.

**Key Benefits:**

- Improved Code Quality: TDD ensures code reliability, quality, and maintainability.

- Early Bug Detection: TDD helps detect bugs early in the development process.

- Constant Feedback: TDD provides constant feedback about code functionality.

- Better Design: TDD promotes better design and architecture.

**TDD Process:**

1. Red: Write a test and run it to see it fail.

2. Green: Write code to make the test pass.

3. Refactor: Refactor code to remove duplication and improve design.

**Approaches:**

- Inside-Out: Focuses on testing individual units of code.

- Outside-In: Focuses on testing user behavior and interactions.

**Advantages:**

- Improved code quality

- Early bug detection

- Constant feedback

- Better design

**Disadvantages:**

- Increased code volume

- Maintenance overheads

- Time-consuming test processes

- Testing environment set-up

**Conclusion:**

TDD is a valuable approach for developing high-quality software. It promotes early bug detection, improves code quality, and aligns code with business requirements. While it has its challenges, the benefits of TDD make it a worthwhile methodology to adopt.

***Unit Testing - Software Testing***

Unit testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure they perform as expected.

**Key Benefits:**

- Early Detection of Issues: Unit testing helps detect and fix issues early in the development process.

- Improved Code Quality: Unit testing ensures each unit of code works as intended and meets requirements.

- Increased Confidence: Unit testing provides developers with confidence in their code.

- Faster Development: Unit testing enables developers to work faster and more efficiently.

**Types of Unit Testing:**

- Manual Unit Testing: Manual testing involves checking each part of a project by hand.

- Automated Unit Testing: Automated testing uses special tools to run tests automatically.

**Unit Testing Techniques:**

- Black Box Testing: Testing input, user interface, and output parts.

- White Box Testing: Testing functional behavior and internal design structure.

- Gray Box Testing: Testing code performance and executing test cases.

**Unit Testing Tools:**

- Lambda Test: Cloud-based platform for cross-browser testing.

- JUnit: Java testing framework for creating and running unit tests.

- N Unit: .NET framework for unit testing C# applications.

**Best Practices:**

- Use a Unit Test Framework: Use testing frameworks to automate and standardize testing.

- Automate Unit Testing: Automate unit tests to run during key development events.

- Assert Once: Each unit test should have only one assert statement.

**Conclusion:**

Unit testing is crucial for ensuring software quality and longevity. While it requires significant time and effort, its benefits make it a worthwhile investment. By following best practices and using the right tools, developers can write effective unit tests that improve code quality and reduce bugs.

***Introduction to JUnit 5***

**What is JUnit 5?**

JUnit 5 is the latest version of the JUnit testing framework, designed for writing and running tests in Java applications.

**Components:**

- JUnit Platform: Provides a launching mechanism for testing frameworks on the JVM.

- JUnit Jupiter: Introduces new programming techniques for developing test cases.

- JUnit Vintage: Allows running JUnit 3 and JUnit 4 test cases on the JUnit 5 platform.

**Key Features:**

- Annotations: @Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, @DisplayName, @Disabled.

- Test Life Cycle Methods: @BeforeEach, @AfterEach, @BeforeAll, @AfterAll.

- Assertions: Methods for checking expected results, such as assertEquals, assertTrue, assertFalse.

- Assumptions: Conditionally skip tests based on preconditions.

- Parameterized Tests: Run tests with different parameters using @ParameterizedTest.

- Dynamic Tests: Generate tests at runtime using @TestFactory.

- Tagging and Filtering: Categorize and filter tests using @Tag.

**Benefits:**

- Improved Test Writing: JUnit 5 provides more flexible and powerful features for writing tests.

- Better Test Organization: Annotations and test life cycle methods help organize tests effectively.

- Efficient Test Execution: Run tests in various ways, including IDE integration, build tools, and console launcher.

**Conclusion:**

JUnit 5 is a powerful testing framework that offers many features and improvements over JUnit 4. Its components, annotations, and features make it an ideal choice for writing and running tests in Java applications.

**What is Unit Testing?**

Unit testing is a fundamental part of software development that ensures individual components of a program work effectively as expected.

**10 Best Practices for Unit Testing:**

1. Planning a Test: Plan and allocate resources for unit testing to ensure smooth development.

2. Writing a Clean and Readable Test: Write tests that are easy to understand and maintain.

3. Use AAA Pattern: Use the Arrange, Act, and Assert pattern to make tests more readable.

4. Deterministic Tests: Ensure tests are deterministic and avoid non-deterministic tests.

5. Avoid Logic in Tests: Avoid using logic in tests to make them more readable and deterministic.

6. Test Coverage: Aim for high test coverage to detect more problems.

7. Automated Test: Use automated testing to detect bugs early and save resources.

8. Write Tests as the Project Develops: Write tests during development to understand product code better.

9. One Use Case per Unit Test: Use one case per unit test to make tests more definitive and easier to manage.

10. Test Documentation: Keep test documentation to review, repeat, and archive tests.

**Benefits of Unit Testing:**

- Early Bug Detection: Unit tests enable earlier bug detection and resolution.

- Ensures Safety: Unit tests serve as a safety net for developers, preventing regressions.

- Code Quality: Unit tests contribute to enhanced code quality and better design practices.

- Good Documentation: Unit tests function as a form of documentation, demonstrating how code is intended to be used.

**Conclusion:**

Unit testing is a necessity in software development, and following these best practices can ensure that software is safe, reliable, and efficient. By incorporating unit testing into the development process, developers can detect bugs early, improve code quality, and reduce the risk of regressions.

**What is Mockito?**

Mockito is a popular open-source testing framework for Java that allows you to create mock objects, stub methods, and verify interactions between objects.

**Key Features:**

- Mock Object Creation: Create mock objects to isolate dependencies and test classes in isolation.

- Stubbing: Stub methods to return specific values or throw exceptions.

- Verification: Verify interactions between objects, including method calls and argument values.

- Matchers: Use matchers to specify argument values or types.

**Mockito Tutorials:**

- Getting Started with Mockito: Learn the basics of Mockito, including mock object creation, stubbing, and verification.

- Mocks, Spies, and Partial Mocks: Understand the differences between mocks, spies, and partial mocks.

- Verification: Learn how to verify interactions between objects.

- Hamcrest Matchers: Use Hamcrest matchers to specify argument values or types.

**Mockito Examples:**

- Mockito Hello World Example: A simple example of using Mockito to create a mock object and stub a method.

- Mockito Mock Database Connection Example: An example of using Mockito to mock a database connection.

- Mockito List Matcher Example: An example of using Mockito matchers for lists.

**Mockito Integrations:**

- Powermock - Mockito Integration Example: An example of integrating Powermock with Mockito.

- Mockito Mock Static Method Example: An example of using Powermockito to mock static methods.

- Spring Test Mock Example: An example of using Mockito to mock Spring components.

**Conclusion:**

Mockito is a powerful testing framework that allows you to create mock objects, stub methods, and verify interactions between objects. Its features and integrations make it a popular choice for unit testing in Java.

**Testing in Spring Boot**

**Types of Testing:**

- Unit Testing: Test individual components or units of code in isolation.

- Integration Testing: Test interactions between multiple components or modules.

- End-to-End (E2E) Testing: Test the entire application from end to end, simulating real user scenarios.

**Key Annotations:**

- @SpringBootTest: Indicates a Spring Boot test, loads the complete application context.

- @Run With (Spring Runner.class): Specifies the class to run the tests.

- @MockBean: Mocks a Spring Bean for testing purposes.

- @Test: Denotes a test method.

Testing Dependencies:

- Spring Boot Starter Test: Includes JUnit, Mockito, and other testing libraries.

**Examples:**

- @SpringBootTest: Integration testing, loads the complete application context.

- @WebMvcTest: Unit testing, focuses on MVC components.

- @DataJpaTest: Integration testing, focuses on JPA components.

Additional Annotations:

- @AutoConfigureMockMvc: Automatically configures a MockMvc instance.

- @DirtiesContext: Indicates that the ApplicationContext is dirty and should be closed after the test.

- @Transactional: Used to indicate that a test-managed transaction should be used.

**Conclusion:**

Spring Boot provides a robust testing framework that supports unit testing, integration testing, and end-to-end testing. By using the appropriate testing annotations and strategies, developers can ensure the reliability and correctness of their Spring Boot applications.

**Mocking Repositories and DAOs with Mockito**

Mockito is a powerful mocking framework for Java that simplifies testing database interactions by allowing developers to mock repositories and Data Access Objects (DAOs). This approach ensures fast and reliable unit tests without needing actual database connections.

**Benefits of Mocking**

- Speed and Efficiency: Eliminates the need to interact with a real database, speeding up the testing process.

- Isolation: Tests focus on business logic without being affected by database state or availability.

- Reproducibility: Mocked responses are consistent, ensuring stable and predictable test outcomes.

**Mocking Repositories and DAOs**

- @Mock: Creates a mock instance of the repository or DAO.

- @InjectMocks: Injects the mock into the service being tested.

- when(): Configures mock behavior for specific method calls.

- verify(): Ensures the mocked method was called as expected.

**Best Practices**

- Mock Only What You Own: Focus on mocking the application’s repository or DAO layers.

- Avoid Over-Mocking: Mock only necessary methods.

- Use @InjectMocks Smartly: Automate dependency injection in tests.

- Combine with Integration Tests: Validate real-world scenarios.

- Leverage Argument Captors: Verify interactions with mocked objects effectively.

Example

@ExtendWith(MockitoExtension.class)

class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

void testFindUserByEmail() {

// Arrange

String email = "test@example.com";

User mockUser = new User(1L, "Test User", email);

when(userRepository.findByEmail(email)).thenReturn(Optional.of(mockUser));

// Act

Optional<User> result = userService.findUserByEmail(email);

// Assert

assertTrue(result.isPresent());

assertEquals(email, result.get().getEmail());

verify(userRepository).findByEmail(email);

}

}

**Automation Testing - Software Testing**

**What is Automation Testing?**

Automation testing is a technique that uses special software to perform tasks that people usually do when checking and testing a software product. It automates repetitive tasks, increases test coverage, and ensures the quality of the software.

**Benefits of Automation Testing:**

- Faster Execution: Automation testing is faster than manual testing, allowing for quicker feedback and deployment.

- Increased Test Coverage: Automation testing can cover more test cases, ensuring that the software is thoroughly tested.

- Improved Quality: Automation testing reduces human error, ensuring that the software meets the required standards.

- Cost-Effective: Although the initial investment is high, automation testing is cost-effective in the long run.

**Types of Tests to Automate:**

- Unit Tests: Test individual units of code, such as functions or methods.

- Integration Tests: Test the interactions between different components or modules.

- End-to-End Tests: Test the entire application, from start to finish.

- Performance Tests: Test the application's performance under various loads.

Types of Tests to Do Manually:

- Exploratory Testing: Test the software without pre-defined test cases.

- Visual Regression Testing: Test the visual aspects of the application.

- Usability Testing: Test the user-friendliness of the application.

**Automation Testing Process:**

- Test Tool Selection: Choose the right automation tool for the project.

- Define Scope of Automation: Determine what tests to automate.

- Planning, Design, and Development: Plan and develop the automation tests.

- Test Execution: Execute the automation tests.

- Maintenance: Maintain the automation tests and update them as needed.

**Best Practices for Test Automation:**

- Plan Self-Contained Test Cases: Ensure that test cases are independent and self-contained.

- Use Tools with Automatic Scheduling: Use tools that can schedule tests automatically.

- Set Up Alarms for Test Failure: Set up alarms for test failures to ensure prompt action.

**Popular Automation Tools:**

- Selenium: An open-source tool for web automation testing.

- QTP: A commercial tool for functional and regression testing.

- Appium: An open-source tool for mobile automation testing.

**Advantages and Disadvantages:**

- Advantages: Simplifies test case execution, improves reliability, increases test coverage, and saves time and money.

- Disadvantages: High initial cost, not possible to automate all testing types, and requires programming knowledge.

***SLF4J Tutorial***

**What is SLF4J?**

- SLF4J (Simple Logging Facade for Java) is a logging facade that provides a simple and unified logging interface for Java applications.

- It allows developers to use various logging frameworks (e.g., Logback, Log4j) without being tied to a specific implementation.

**Key Features:**

- Logging Facade: SLF4J acts as a facade, decoupling the logging API from the underlying logging implementation.

- Pluggable Architecture: Supports multiple logging frameworks, making it easy to switch between different logging implementations.

- Parameterized Logging: Supports parameterized logging, which improves performance by avoiding unnecessary string concatenation.

**Benefits:**

- Flexibility: SLF4J provides flexibility in choosing the underlying logging framework.

- Portability: Applications using SLF4J can be easily ported to different logging frameworks.

- Improved Performance: Parameterized logging helps improve performance.

Common Use Cases:

- Logging in Java Applications: SLF4J is widely used in Java applications for logging purposes.

- Integration with Logging Frameworks: SLF4J can be integrated with popular logging frameworks like Logback and Log4j.

Example Usage:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class MyClass {

private static final Logger logger = LoggerFactory.getLogger(MyClass.class);

public void doSomething() {

logger.info("Doing something");

logger.debug("Debug message with parameter: {}", "parameterValue");

}

}

By using SLF4J, developers can write logging code that is independent of the underlying logging framework, making it easier to switch between differe