Week 4 – Errors in Java:

Day 1 - Compile Time/Syntax Errors:

**Compiler Role:**

* Translates Java code (human-readable) into Java bytecode (computer-readable).
* Analogous to a translator converting languages.

**Error Cause:**

* Compiler errors occur when the code breaks language rules (syntax).
* Equivalent to a translator unable to translate an unknown word.

**Nature of Errors:**

* Commonly due to forgotten semicolons, misspelled variables, or rule violations (e.g., using a non-static variable in a static function).

**Example:**

* Incomplete line causing a compile error (missing semicolon at the end).
* Understanding Java syntax is crucial to rectify such errors.

**Identification:**

* Many Java Integrated Development Environments (IDEs) detect compile errors during coding.
* Helps in easily recognizing and rectifying errors as you write code.

Runtime Errors:

**Runtime Error Definition:**

* Occurs during program execution, disrupting the normal flow and leading to abnormal termination.
* Reasons include invalid user input, inaccessible files, lost network connections, or JVM memory exhaustion.

**Scenarios for Runtime Errors:**

* Invalid user input, missing files, lost network connections, and resource failures can cause runtime errors.
* These errors stem from user, programmer, or physical resource issues.

**Detection of Runtime Errors:**

* Not detected by the compiler; they occur during program execution.

**Example and Output Interpretation:**

* Example error: **java.lang.ArrayIndexOutOfBoundsException: 5.**

**Interpretation:**

* Type of error **(ArrayIndexOutOfBoundsException).**
* Information about the error (attempting to access the 5th index of an array).
* Stack trace specifying the class, function, and line number where the error occurred.

**Exception Handling:**

* Aim: Prevent devastating effects of runtime errors and maintain the application's normal flow.
* Methods:

**Java try block:**

* + Encloses code that might throw an exception.
  + Should be followed by catch or finally block.

**Java catch block:**

* + Handles exceptions by declaring the type of exception within the parameter.
  + Multiple catch blocks can be used with a single try block.

**Java finally block:**

* + Executes important code (e.g., closing connections or streams).
  + Always runs, whether an exception is handled or not.

Day 2 – Logic Errors:

**Logic Errors:**

* Logic errors occur when code compiles and runs without exceptions but fails to produce the expected output due to programmer assumptions, typos, or flawed logic.
* Identification is challenging during coding; testing by comparing expected versus actual results helps detect these errors.
* Logic errors might remain unnoticed, leading to flawed applications being deployed in production.
* Writing error-free code that compiles and runs doesn't guarantee the absence of logic errors; they are common occurrences in programming.

**Debugging:**

* Debugging is a systematic process of finding and fixing bugs or defects in a computer program.
* Bugs arise when something assumed to be right turned out to be wrong, making the process challenging.
* Vital in identifying and resolving errors, fundamental in a programmer's daily work.
* Helps maintain the quality and functionality of software applications.

**Debugging Process:**

1. **Localizing a Bug:**

* Identifying the bug's origin is crucial before attempting to fix it.
* Errors can be deceptive, making pinpointing them challenging.

1. **Classifying the Error:**

* Categorizing errors (compile, runtime, or logic) aids in effective solutions.
* Failure to classify correctly can hinder fixing the error.

1. **Understanding an Error:**

* Complete understanding of the error is necessary before fixing it.
* Avoids inadvertently causing more issues within the codebase.

1. **Repairing an Error:**

* Fixing the error involves more than code modification.
* Proper documentation of fixes is essential, aiding in future reference and learning.

Debugging Techniques:

1. **Exploiting Compiler Features:**

* Utilizing the Java compiler's static analysis capabilities to detect syntax or semantic issues before execution.

1. **The abused println() Debugging Technique:**

* Involves inserting print statements to track code execution flow and data values during runtime.
* Considered ad-hoc, time-consuming, and not reusable.

1. **Logging:**

* Recording information messages or events to monitor program status and diagnose issues.
* Implemented through tools like log4j, offering various logging levels.

1. **Defensive Programming and Assertions:**

* Using assertions to validate code assumptions at specific points; helpful in identifying code problems.

1. **ACI Debugging Technique:**

* Explaining code to someone else to rethink assumptions and solve problems effectively.

1. **Reading the Code Through:**

* Reviewing code away from the terminal to understand its logic and identify issues.

1. **The Debugger:**

* An interactive tool allowing line-by-line code execution inspection, variable inspection, and breakpoints setting.
* Useful when other methods fail to identify problems, providing detailed control over code execution.

Day 3 – Common Errors in Java:

**“… Expected”**

* Missing semicolon or closing parenthesis leads to this error.
* Ensure balanced parentheses and check the previous line.

**“Unclosed String Literal”**

* Occurs when a string literal lack closing quotation marks.
* Correct by adding the needed quote marks or breaking long literals.

**“Illegal Start of an Expression”**

* Less-helpful error message caused by syntax mismatch.
* Review statements where the error occurs.

**“Cannot Find Symbol”**

* Arises due to undeclared identifiers or incorrect usage.
* Verify variable declaration, usage scope, and imported classes.

**“Public Class Should Be in File”**

* Class and Java file names mismatch.
* Name both consistently and ensure case consistency.

**“Incompatible Types”**

* Error occurs when types don't match during assignment.
* Convert types or redefine code logic.

**“Invalid Method Declaration”**

* Missing return type in the method signature.
* Specify the method's return type or use "void" for non-return methods.

**“Missing Return Statement”**

* A method returning a value lacks a return statement.
* Ensure all paths of value-returning methods have a return statement.

**“Possible Loss of Precision”**

* Occurs during type conversion causing data loss.
* Explicitly define variable types or perform type conversions.

**“Reached End of File While Parsing”**

* Missing closing curly brace leads to this error.
* Check code indentation and balance braces.

Runtime Errors:

**“ArrayIndexOutOfBoundsException”**

* Accessing an array index out of its range.
* Correct index definitions and loops.

**“StringIndexOutOfBoundsException”**

* Accessing parts of a string beyond its length.
* Check string indexing and lengths.

**“NullPointerException”**

* Attempting to use a null object reference.
* Ensure object references are valid before using them.

**“NoClassDefFoundError”**

* Unable to find a class file with the main method.
* Check file location, naming, and case sensitivity.

**“NoSuchMethodFoundError”**

* Trying to call an undefined method.
* Review method declarations for typos or missing methods.

**“NoSuchProviderException”**

* Requesting an unavailable security provider.
* Check JRE configuration and environment settings.

**“AccessControlException”**

* Denied access to system resources.
* Review permissions and resource access.

**“ArrayStoreException”**

* Violating rules of object casting in arrays.
* Ensure consistent object types in arrays.

**“UnsupportedEncodingException”**

* Unsupported character encoding used.
* Verify encoding support in the Java Virtual Machine.

**“TimeoutException”**

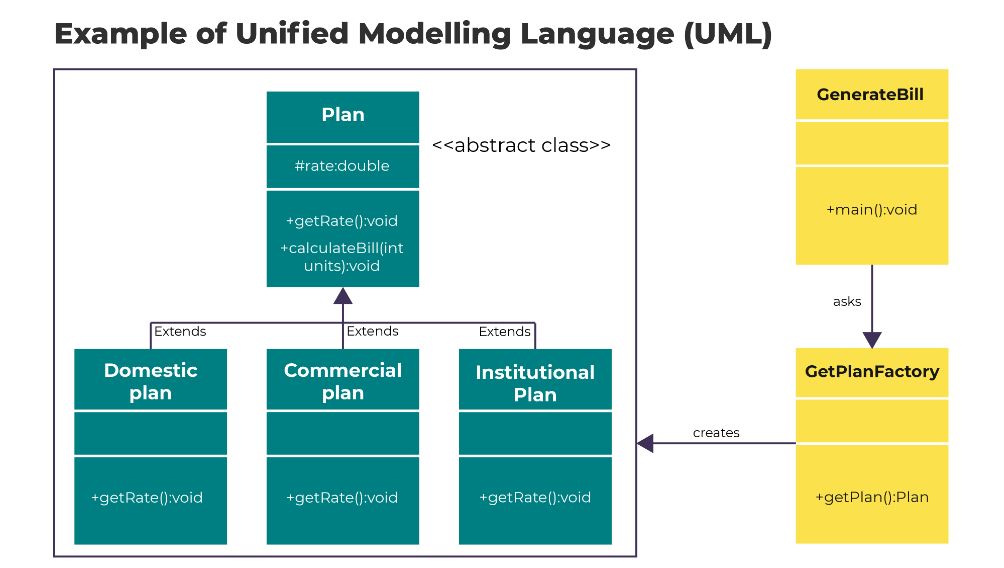
* Blocking operation times out.
* Review the code and handling of blocking operations for timeout scenarios.

Factory Method Patterns:

* The factory method pattern involves defining an interface or abstract class for creating objects, allowing subclasses to determine the class used for object creation.

**This pattern is useful when:**

1. A class is unaware of which subclasses need to be created.
2. Subclasses specify the objects to be created.
3. Parent classes determine object creation for subclasses.



Day 4 – Code Refactoring:

**What is refactoring?**

* Refactoring is a vital technique used in software development to enhance existing code without altering its observable behaviour.
* The primary aim of refactoring is to improve code quality, making it more maintainable, readable, and efficient.
* It's a fundamental Agile practice that allows developers to:
  1. Improve code maintainability, reducing costs associated with software maintenance.
  2. Enable efficient introduction of new requirements without introducing bugs.
  3. Restructure code to align with design patterns or best practices.

**Importance of Refactoring:**

* As software grows and complexity, bugs often infiltrate the codebase, decreasing code reliability.
* Refactoring plays a pivotal role in addressing these issues by making the code more understandable and maintainable.
* This helps in reducing costs and freeing up development resources for other tasks.
* Additionally, well-structured code makes it easier to introduce new features seamlessly and with minimal issues.

**The Refactoring Process:**

* Refactoring involves making incremental, logical changes to code structure while ensuring its behaviour remains unchanged.
* By breaking down changes into small steps and running tests after each step, potential bugs introduced can be identified immediately, facilitating quick resolution.
* This incremental approach ensures better control over the code changes and minimizes the risk of introducing errors.

**Phases of Refactoring:**

1. **Detecting a Problem:** Identify issues or areas within the code that need improvement.
2. **Characterizing the Problem:** Understand the reasons behind the necessary changes, assessing the benefits and potential risks.
3. **Designing a Solution:** Define the desired state of the code and plan the transformations needed to achieve it.
4. A diagram of a problem

   Description automatically generated**Modifying the Code:** Implement code transformations in incremental steps while ensuring the code functions as expected throughout the process.

**When is Refactoring Used?**

1. **Code Maintenance:** To address codebase issues, such as eliminating code smells or improving readability.
2. **Feature Enhancement:** When introducing new features that may conflict with existing code or design.
3. **Code Restructuring:** Aligning code with established design patterns or best practices.

**Examples of Refactoring:**

1. **Renaming:**

* **Purpose:** Correct misleading or confusing names of methods, variables, classes, etc.
* **Process:** Update all references to the renamed entity across the codebase.
* **Impact:** Renaming may require adjustments in subclasses, clients, file locations, and directories, along with updates in the version control system.

1. **Moving a Class:**

* **Purpose:** Relocate a class from one package to another where it fits more appropriately.
* **Process:** Update all import statements and references to the class in its new package.
* **Impact:** Involves moving the file to the new location and updating references in the source control system.

1. **Extract Method:**

* **Purpose:** Break down lengthy methods to enhance code readability and maintainability.
* **Process:** Identify a section of code performing a specific logical task and replace it with a call to a new method.
* **Impact:** Improves readability by segregating logical units of work into separate methods, making the code more understandable.

**4. Extracting a Superclass:**

* **Purpose:** Introduce an abstract class as a parent to an existing class to manage common functionality.
* **Process:** Pull up common behaviour from the existing class into the new abstract parent class.
* **Impact:** Clients of the original class are modified to reference the new parent class, enabling different implementations through polymorphism.