## Compile Time/Syntax Errors

**Compiler Errors/Syntax Errors**

Occur during code compilation, indicating mistakes that prevent the program from being created. These need to be fixed before running the program.

Most modern Java IDEs detect these errors as you type, helping you identify and fix them easily while writing code.

**Class BadExample {**

**Public static void main(String[] args) {**

**System.out.println(“Happy Coding!”)**

**}**

**}**

**Runtime Errors**

Occur when the program is running and encounters an issue that wasn't anticipated, causing it to crash or behave unexpectedly.

**Causes:** Various scenarios like invalid user input, missing files, lost network connections, or running out of memory.

For example, accessing an invalid array index in Java code such as **java.lang.ArrayIndexOutOfBoundsException**

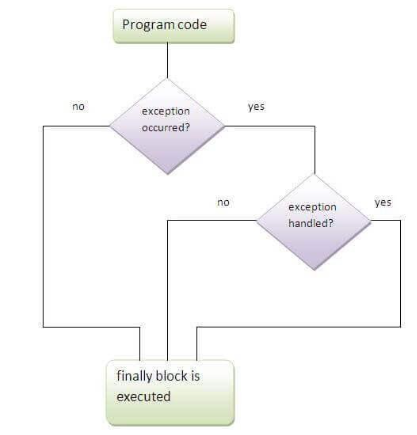
The error message includes details like the type of error, specific details about the error (e.g., array index attempted), and a stack trace indicating where the error occurred in the code.

Exception Handling

Purpose: To handle runtime errors so that they don't lead to program crashes.

In Java, handled using try, catch, and finally blocks.

* **Try block:** Contains code that might throw an exception. Followed by catch or finally blocks.
* **Catch block:** Handles exceptions by declaring the type of exception it can handle.
* **Finally block:** Executes essential code, like closing connections, always runs whether an exception is handled or not.



**try {**

// code that may throw an exception

**} catch (Exception\_class\_Name ref) {**

**finally {**

**}**

**}**

<https://www.youtube.com/watch?v=JTjeGpSUL2M>

## Logic Errors

These errors don't crash the program or produce error messages, but they cause the program to function incorrectly due to flawed logic or reasoning in the code.

Not easily recognizable during compilation or runtime; testing and comparing expected versus actual results help uncover these errors.

Debugging

Process of identifying and reducing bugs or defects in a computer program. **Cause of Errors:** Stem from something believed to be right but was actually wrong, making it challenging to pinpoint and rectify these issues.

Steps to debugging:

1. **Localizing a Bug:** Identifying the source of the error in the code.
2. **Classifying the Error:** Categorizing the error (compile, runtime, logic) aids in resolution.
3. **Understanding the Error:** Fully comprehending the error before attempting to fix it to prevent further issues.
4. **Repairing the Error:** Modifying code to fix the error, documenting fixes, and learning from mistakes.

General Debugging Techniques:

1. **Exploiting Compiler Features:** Utilizing static code analysis to detect basic semantic problems or syntax errors.
2. **Println() Debugging Technique:** Adding print statements in code to track control flow and data values, despite its drawbacks like temporary and ad-hoc nature.
3. **Logging:** Recording information messages or events automatically to monitor program status and diagnose problems.
4. **Defensive Programming and Assertions:** Using expressions that should evaluate as true at a specific point in code to detect problems.
5. **ACI Debugging Technique:** Explaining code to someone else to rethink assumptions and problem-solving.
6. **Reading the Code Through:** Reviewing code outside the programming environment to understand its workings without execution.
7. **Using a Debugger:** Employing a tool that allows line-by-line code execution, variable inspection, and program flow control to detect and solve issues, especially when other methods fail.

<https://www.youtube.com/watch?v=-GkRuFU_sUg>

<https://www.youtube.com/watch?v=i0o60sH_AVw>

## List of Errors

Compiler Errors

* **"Expected" Error:** Often due to missing semicolons or closing parentheses. Checking for correct parenthesis balance and looking at the preceding line might resolve it.
* **"Unclosed String Literal":** Missing quotation marks at the end of a string literal or an extended string beyond a line. Correct by adding missing quotes or breaking the string into multiple literals.
* **"Illegal Start of an Expression":** Vague error caused by syntax mismatch.
* **"Cannot Find Symbol":** Arises when identifiers are not declared or misspelled.
* **"Public Class Should Be in File":** Class name mismatch with the file name. Resolved by ensuring consistent naming and casing.
* **"Incompatible Types":** Occurs during incompatible variable assignments.
* **"Invalid Method Declaration; Return Type Required":** Absence of a stated return type in method signature.
* **"Missing Return Statement":** Occurs when a method lacks a return statement for a non-void type.
* **"Possible Loss of Precision":** Error due to assigning more information to a variable than it can hold.
* **"Reached End of File While Parsing":** Missing closing curly brace causing parsing issues.
* **"Unreachable Statement":** Placed after a break or return statement, preventing execution.
* **"Variable Might Not Have Been Initialized":** Local variable within a method not initialized.
* **"Operator Cannot be Applied to Type":** Inappropriate usage of operators for specific types.
* **"Inconvertible Types":** Illegal conversions attempted, e.g., boolean to integer.
* **"Missing Return Value":** Incorrect type in the return statement.
* **"Cannot Return a Value From Method Whose Result Type Is Void":** Void method attempts to return a value.
* **"Non-Static Variable Cannot Be Referenced From a Static Context":** Issue accessing non-static variables from a static method.
* **"Non-Static Method Cannot Be Referenced From a Static Context":** Occurs when calling a non-static method from a static context.
* **"(Array) Not Initialized":** Error when declaring but not initializing an array.

Runtime Errors

* **"ArrayIndexOutOfBoundsException":** Accessing an array index beyond its bounds.
* **"StringIndexOutOfBoundsException":** Attempting to access string indexes beyond its bounds.
* **"NullPointerException":** Using an object reference without a value.
* **"NoClassDefFoundError":** Interpreter can't find the file containing a class with the main method.
* **"NoSuchMethodFoundError":** Occurs when the method definition is missing or contains a typo.
* **"NoSuchProviderException":** Requested security provider is not available.
* **"AccessControlException":** Denied access to system resources.
* **"ArrayStoreException":** Violation of casting elements in Java arrays.
* **"Bad Magic Number":** Class definition file issues, including incorrect file format.
* **"Broken Pipe":** Data stream closure issues.
* **"Could Not Create Java Virtual Machine":** Invocation of Java with incorrect arguments.
* **"Class File Contains Wrong Class":** Mismatch between class name and file or incorrect package statement.
* **"ClassCastException":** Incorrect object cast to a different class.
* **"ClassFormatError":** Issues reading or interpreting a class file.
* **"ClassNotFoundException":** Run-time missing class, common when file or class name mismatches.
* **"ExceptionInInitializerError":** Static initialization error.
* **"IllegalBlockSizeException":** Decryption length mismatch.
* **"BadPaddingException":** Encryption padding error.
* **"IncompatibleClassChangeError":** Base class changes affecting compiled child class.
* **"FileNotFoundException":** Non-existent file specified by the pathname.
* **"EOFException":** End of file or stream reached unexpectedly during input.
* **"UnsupportedEncodingException":** Unsupported character encoding used.
* **"SocketException":** Error accessing or creating a socket.
* **"SSLException":** Failure in SSL-related operations.
* **"MissingResourceException":** Missing resource, often due to properties file misconfiguration.
* **"NoInitialContextException":** Issues creating a connection for naming operations.
* **"NoSuchElementException":** Iteration tries to access a nonexistent element.
* **"NoSuchFieldError":** Accessing a nonexistent field in an object.
* **"NumberFormatException":** Invalid conversion of string to numeric type.
* **"TimeoutException":** Blocking operation times out.

## Design Patterns

<https://www.youtube.com/watch?v=rFqf7NduU8w>

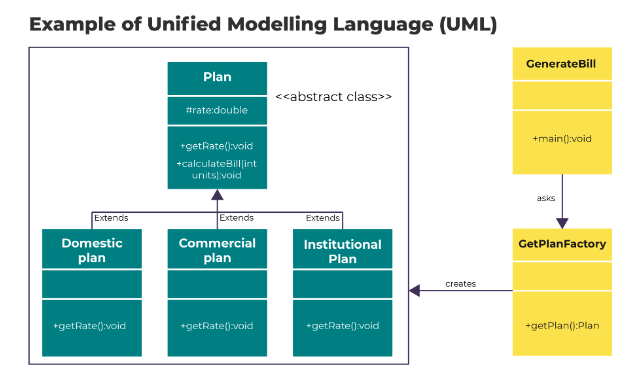
<https://www.journaldev.com/1827/java-design-patterns-example-tutorial>

Factory Method Patterns

A factory pattern involves an interface or abstract class to create objects, letting subclasses determine the class to instantiate. It's useful when:

* A class doesn't know the required subclasses.
* Subclasses specify the objects to create.
* Parent classes choose object creation for their subclasses.

Example using Unified Modeling Language (UML)



Creation of an abstract class "Plan" and concrete classes extending it.

Definition of "GetPlanFactory" as a factory class.

Usage of "GetPlanFactory" by "GenerateBill" to acquire a Plan object based on provided information (e.g., DOMESTICPLAN, COMMERCIALPLAN, INSTITUTIONALPLAN).

## Refactoring

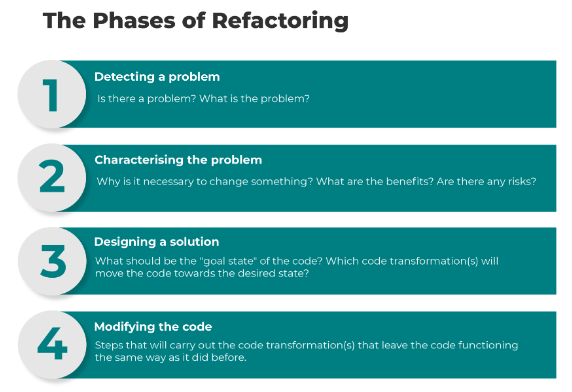
Refactoring

Refactoring is an Agile technique employed by developers to enhance existing software. As software grows, complexity and bugs increase, reducing code reliability and making it harder to maintain and extend.

**Refactoring involves:**

* Improving existing code design without altering its observable behavior.
* Making small, logical changes to the code structure.
* Ensuring each change doesn't introduce bugs by testing after every step.
* Allowing for easier maintenance and introducing new requirements more efficiently.

This technique resolves conflicts between maintaining and enhancing software, aiming to reduce the effort needed for modifications and aiding non-destructive changes to the source code's structure.



A refactoring operation proceeds roughly via the following phases:

* **Detecting a problem** – Is there a problem? What is the problem?
* **Characterising the problem** – Why is it necessary to change something? What are the benefits? Are there any risks?
* **Designing a solution** – What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state?
* **Modifying the code** – Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before.

When to Consider Refactoring:

**Continuous Improvement:** Ideally, refactoring should be part of an ongoing quality improvement process for all software developers.

**Bug Fixing or Extension:** Refactoring becomes useful when fixing bugs or extending code.

**Integration with Other Activities:** Refactoring alongside maintenance or adding features makes it more acceptable, as it doesn't require an additional testing phase.

**Maintaining Observable Behavior:** Refactoring must ensure that the external behavior of the code remains unchanged.

**Conditions for Manual Refactoring:** Manual refactoring is practical when the system can be rebuilt quickly and there are automated regression tests available for frequent use.

Examples of refractoring includes:

1. **Renaming:** Changing the name of a variable, method, class, or any other element to make its purpose or usage clearer.
2. **Moving a Class:** Reorganizing the structure by relocating a class to a different package or module for better organization or to improve modularity.
3. **Extract Method:** Taking a section of code within a method and creating a new method to perform that specific task. This helps in making the code more readable and reusable.
4. **Extracting a Superclass:** Identifying common functionalities within different classes and creating a new superclass to contain these shared features, promoting code reusability and reducing redundancy.