1. compiled language vs interpreted language

Compile one time and run it everywhere

Multi-thread: handle all requests concurrently

Java is unique in that it is both **compiled** and **interpreted**, depending on the stage of execution. Here's a comparison of compiled and interpreted languages, with an explanation of how Java fits into each category:

Compiled Language

- **Definition**: A compiled language translates the entire source code into machine code (binary) before execution. This machine code is directly executed by the computer's CPU.
- **Key Characteristics**:
    - Faster runtime performance since code is already converted to machine language.
    - Errors are caught during the compilation phase.
    - Examples: C, C++, Rust.

Java as a Compiled Language:

- Java source code (.java) is compiled by the **Java Compiler (javac)** into **bytecode** (.class files).
- Bytecode is not machine code but a platform-independent intermediate representation.
- Compilation happens only once, producing the bytecode.

Interpreted Language

- **Definition**: An interpreted language translates and executes code line by line at runtime, typically through an interpreter.
- **Key Characteristics**:
    - Slower than compiled languages because translation happens during execution.

- Easier to debug during development due to real-time execution.
- Examples: Python, JavaScript.

Java as an Interpreted Language:

- Java bytecode (.class files) is interpreted by the **Java Virtual Machine (JVM)**.
- The JVM translates bytecode into machine code for the host platform and executes it. This is why Java is considered "Write Once, Run Anywhere" (WORA).

Hybrid Execution in Java

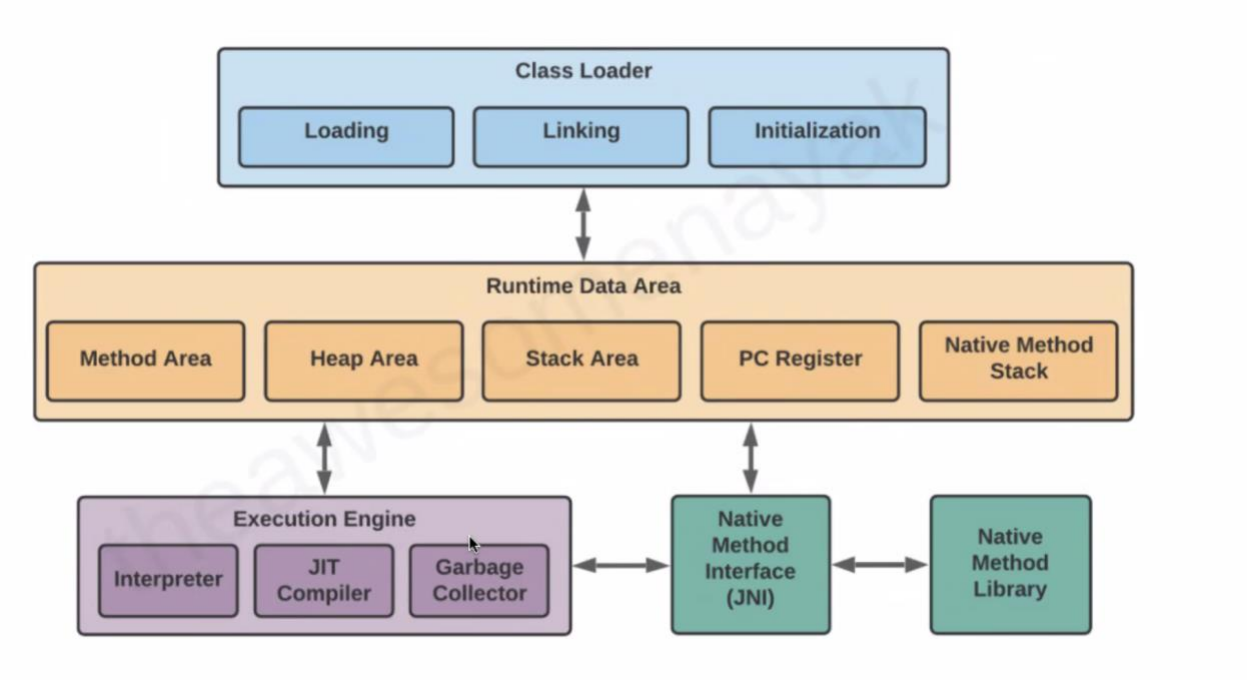
Java's execution combines both approaches:

1. **Compilation Phase**:
   - **Java source code → Compiled by javac → Bytecode.**
2. **Interpretation and Just-In-Time (JIT) Compilation**:
   - **The JVM interprets bytecode line by line.**
   - **To improve performance, the JVM employs a Just-In-Time (JIT) Compiler to compile frequently executed bytecode into native machine code during runtime.**

Key Advantages of Java's Approach:

- **Platform Independence**: Bytecode can run on any machine with a compatible JVM.
- **Performance Optimization**: JIT compilation boosts runtime performance.
- **Development Flexibility**: Combines the benefits of static (compiled) and dynamic (interpreted) execution.

In summary, Java is a **compiled language** because of its initial compilation into bytecode, but it is also an **interpreted language** because the JVM executes the bytecode.



The **Java Virtual Machine (JVM)** is the runtime environment for executing Java applications. It provides platform independence by interpreting Java bytecode into native machine code specific to the host system. Here's an overview of the JVM architecture:

https://www.freecodecamp.org/news/jvm-tutorial-java-virtual-machine-architecture-explained-for-beginners/

JVM Architecture Components

1. **Classloader Subsystem**

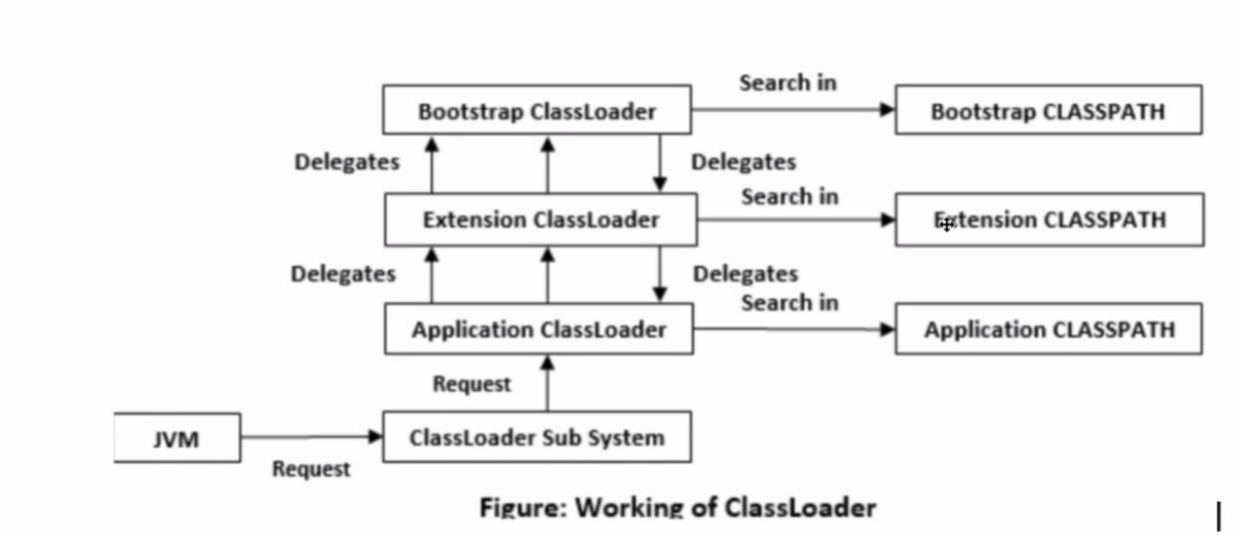


Figure: Working of ClassLoader

2.

Bootstrap/extension/application class extension

The phrase **"Bootstrap/extension/application class extension"** refers to the hierarchical structure of class loading in the JVM. This is an integral part of the **Classloader Subsystem**, which loads classes into memory during the execution of a Java program.

Classloader Hierarchy in JVM

1. **Bootstrap ClassLoader**
    - **Purpose: Loads the core Java classes from the rt.jar file or the Java standard libraries.**
    - **Location: This is the top-most classloader and is written in native code (C or C++).**
    - **Classes Loaded:**
        - **Core Java classes like java.lang.*, java.util.*, java.io.*.**
2. **Extension ClassLoader**
    - **Purpose: Loads classes from the lib/ext directory or any other specified directories using the java.ext.dirs system property.**
    - **Role: Provides additional functionality by loading extension classes (JAR files or classes that extend the standard library).**
    - **Classes Loaded:**
        - **Classes that extend the core Java functionality.**

3. **Application ClassLoader**
    - **Purpose: Loads classes from the application's classpath (-cp or CLASSPATH environment variable).**
    - **Role: Responsible for loading user-defined classes and third-party libraries used in the application.**
    - **Classes Loaded:**
        - **User-defined application classes and libraries (e.g., com.example.myapp.*).**

Class Loading Process

1. **Parent Delegation Model**:
    - **When a classloader receives a request to load a class:**
        - It first delegates the request to its parent classloader.
        - If the parent cannot load the class, the current classloader attempts to load it.
    - **This ensures that the core Java classes are loaded only once by the bootstrap classloader.**
2. **Customization**:
    - **You can create custom classloaders by extending java.lang.ClassLoader to load classes in a non-standard way.**
    - **For example, a custom loader could load classes over a network.**

How It Works Together

- The **Bootstrap ClassLoader** loads foundational classes (like Object and String).
- The **Extension ClassLoader** adds extra functionality (e.g., encryption or database connectivity libraries).
- The **Application ClassLoader** ensures that user-defined and third-party libraries are available for your program.

If you need further clarification or have specific examples, let me know!

- Responsible for loading, linking, and initializing Java classes.
- **Key Steps**:
  - **Loading**: Reads .class files and loads bytecode into memory.
  - **Linking**: Verifies bytecode, prepares memory for static fields, and resolves references.
  - **Initialization**: Executes static initializers and assigns static variables.

3. **Runtime Data Areas** JVM allocates memory into specific areas for program execution:
   - **Method Area: (after java 8)**
     - **Stores class-level data like runtime constant pool, method data, and code for methods.**
     - **Shared among all threads.**
   - **Heap:**
     - **Stores objects and class instances.**
     - **Shared among all threads.**
   - **Stack:**
     - **Thread-specific memory for storing method call information (stack frames).**
     - **Each frame contains local variables, operand stacks, and method return values.**
   - **PC Register:**
     - **Holds the address of the next instruction to be executed in a thread.**
     - **Each thread has its own PC register.**
   - **Native Method Stack:**

        - **Manages calls to native (non-Java) methods, like those written in C/C++.**

4. **Execution Engine** The execution engine interprets or compiles the bytecode for execution: (.java->.class)
    - **Interpreter:**
        - **Reads and executes bytecode instructions one at a time.**
        - **Slower due to repetitive translation.**
    - **Just-In-Time (JIT) Compiler:**
        - **Converts frequently executed bytecode into native machine code for faster performance.**
        - **Optimizations include inlining, loop unrolling, and dead code elimination.**
    - **Garbage Collector (GC):**
        - **Automatically deallocates memory for objects no longer in use.**
        - **Various algorithms, like Mark and Sweep or Generational GC, manage memory effectively.**

The **Garbage Collector (GC)** in Java is a part of the JVM responsible for **automatic memory management**. It identifies and removes objects that are no longer in use to reclaim memory for future allocation, relieving developers of manual memory management.

How Garbage Collection Works in Java

1. **Heap Memory Management**
    - **The heap is divided into:**
        - **Young Generation: Stores short-lived objects (e.g., local variables).**
        - **Old Generation (Tenured): Stores long-lived objects.**
        - **Permanent Generation (deprecated in Java 8+): Previously stored metadata about classes (now in Metaspace).**

- **Objects move between generations as their lifetimes increase.**

2. **Garbage Collection Process**
    - **The GC process involves:**
        - **Mark**: Identifies which objects are still reachable.
        - **Sweep**: Reclaims memory occupied by unreachable objects.
        - **Compact (Optional)**: Rearranges objects in memory to reduce fragmentation.
3. **GC Algorithms**
    - **Java provides multiple garbage collection algorithms, including:**
        - **Serial GC: Single-threaded, suitable for single-threaded applications.**
        - **Parallel GC(default): Multi-threaded, optimized for throughput.**
        - **CMS (Concurrent Mark-Sweep) GC(removed since java 14): Reduces pause times for applications requiring low latency.**
        - **G1 (Garbage First) GC: Splits heap into regions and prioritizes garbage collection in regions with the most garbage.**

Phases of Garbage Collection

1. **Young Generation Collection**:
    - **Also called a Minor GC.**
    - **Collects short-lived objects from the Eden Space and promotes surviving objects to the Survivor Spaces or the Old Generation.**
2. **Old Generation Collection**:
    - **Also called a Major GC.**
    - **Cleans up objects in the old generation, which can cause longer pauses.**

3. **Full GC**:
    - **Involves both young and old generations.**
    - **Typically more time-consuming and should be minimized for performance-critical applications.**

Reachability Analysis

- Java uses **reachability analysis** to determine if an object is still in use.
- Reachable objects:
    - Referenced by **GC Roots**, such as static fields, active threads, or local variables.
- Unreachable objects are marked for collection.

Garbage Collector Types

1. **Serial GC**:
    - **Designed for single-threaded applications.**
    - **Use: -XX:+UseSerialGC.**
2. **Parallel GC**:
    - **Multi-threaded, focuses on throughput.**
    - **Use: -XX:+UseParallelGC.**
3. **CMS GC**:
    - **Concurrent, reduces pause times.**
    - **Use: -XX:+UseConcMarkSweepGC.**
4. **G1 GC**:
    - **Balances throughput and latency.**
    - **Prioritizes regions with the most garbage.**
    - **Use: -XX:+UseG1GC.**
5. **ZGC (Z Garbage Collector)**:

- **Low-latency GC designed for large heaps.**
- **Use: -XX:+UseZGC.**

Tuning GC Performance

- Use JVM options to optimize GC behavior:
  - -Xms and -Xmx: Set initial and maximum heap size.
  - -XX:NewRatio: Adjust ratio between young and old generations.
  - GC logging options:
    - -XX:+PrintGCDetails
    - -Xlog:gc

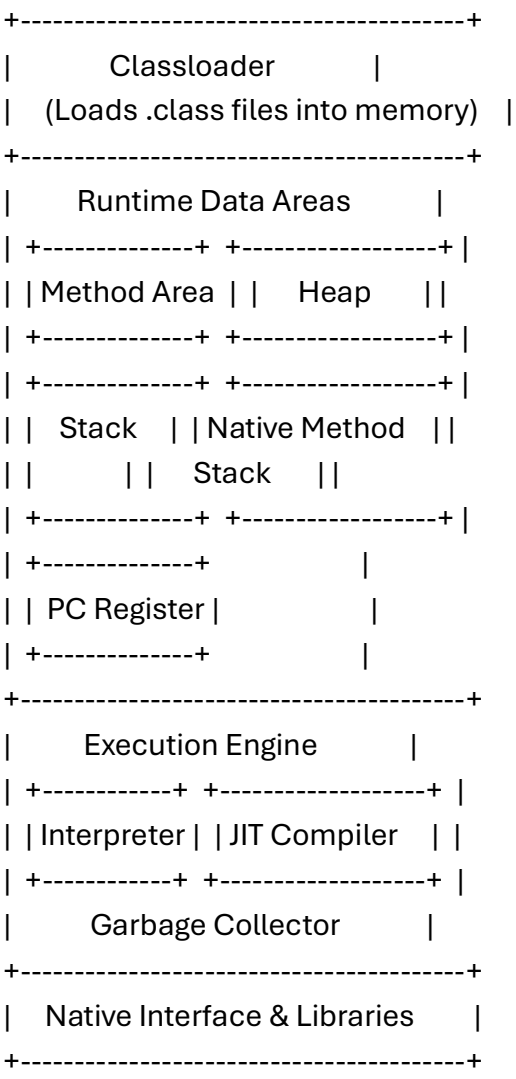
Key Benefits

1. **Automatic Memory Management**: Eliminates manual allocation and deallocation.
2. **Reduces Memory Leaks**: Minimizes risk by reclaiming unused memory.
3. **Improves Developer Productivity**: Focus shifts to application logic rather than memory management.

Let me know if you'd like to explore any GC mechanism in detail!

5. **Native Interface**
   - **Facilitates interaction between the JVM and native libraries (written in C, C++).**
   - **Enables Java applications to use platform-specific functionalities through Java Native Interface (JNI).**
6. **Native Method Libraries**
   - **Dynamic libraries required for the execution of native methods (e.g., .dll, .so files).**

JVM Architecture Diagram



Key Features of JVM Architecture

1. **Platform Independence**: Bytecode runs on any machine with a compatible JVM.
2. **Automatic Memory Management**: The garbage collector reduces the risk of memory leaks.
3. **Thread Management**: JVM supports multithreading and provides isolated memory for each thread.

4. **Performance Optimization**: The JIT compiler enhances runtime efficiency by translating bytecode to native code dynamically.

Let me know if you'd like to dive deeper into any specific component!

Keywords in java

Category	Keywords
Access Modifiers	private, protected, public
Control Flow	if, else, switch, case, default, for, while, do, break, continue, return
Exception Handling	try, catch, finally, throw, throws
Class and Object	class, interface, extends, implements, this, super, new
Primitive Types	byte, short, int, long, float, double, char, boolean, void
Modifiers	final, static, abstract, synchronized, native, transient, volatile, strictfp
Packages and Imports	package, import
Memory Management	new, delete (not used in Java), null
Loop Control	break, continue
Others	assert, enum, instanceof, const, goto (reserved but not used), default, volatile

Keyword	Description
abstract	Specifies that a class or method is incomplete and must be implemented later.
assert	Used for debugging by making an assertion.
boolean	Declares a variable of type boolean (true/false).
break	Exits a loop or switch statement prematurely.
byte	Declares a variable of type byte (8-bit integer).
case	Defines a branch in a switch statement.

catch Handles exceptions in a try-catch block.

char Declares a variable of type character (16-bit Unicode).

class Declares a class.

const Reserved but not used; replaced by final.

continue Skips the current iteration of a loop.

default Specifies the default block of a switch statement.

do Starts a do-while loop.

double Declares a variable of type double (64-bit floating-point).

else Specifies the block of code to execute if the if condition is false.

enum Declares an enumerated type.

extends Indicates inheritance from a parent class.

final Defines constants, prevents inheritance, or method overriding.

finally Ensures execution of a block of code after a try block, regardless of outcome.

float Declares a variable of type float (32-bit floating-point).

for Starts a for loop.

goto Reserved but not implemented.

if Executes a block of code if a condition is true.

implements Indicates that a class implements an interface.

import Imports other Java packages or classes.

instanceof Tests whether an object is an instance of a class.

int Declares a variable of type integer (32-bit).

interface Declares an interface.

long Declares a variable of type long (64-bit integer).

native Indicates that a method is implemented in native code using JNI.

new Creates new objects.

null Represents a null reference.

package Specifies a namespace for classes.

private Declares private access level (accessible within the class only).

protected Declares protected access level (accessible within package and subclasses).

public Declares public access level (accessible from any other class).

return Exits from the current method and optionally returns a value.

short Declares a variable of type short (16-bit integer).

static Indicates that a field or method belongs to the class rather than instances.

strictfp Ensures floating-point calculations are consistent across platforms.

super Refers to the parent class of the current object.

switch Starts a switch statement.

synchronized Prevents thread interference for methods or code blocks.

this Refers to the current object.

throw Used to explicitly throw an exception.

throws Declares the exceptions a method might throw.

transient Indicates that a field should not be serialized.

try Starts a block of code to test for exceptions.

void Specifies that a method does not return a value.

volatile Indicates that a variable's value will be modified by multiple threads.

while Starts a while loop.

Final class, method, variable

Here's a table that clearly differentiates between a **final class**, **final method**, and **final variable** in Java:

Feature	Final Class	Final Method	Final Variable
Definition	A class that cannot be extended (inherited).	A method that cannot be overridden.	A variable whose value cannot be changed once assigned.
Keyword Usage	Declared as final class ClassName { ... }.	Declared as final returnType methodName().	Declared as final dataType variableName.
Inheritance Impact	Prevents the class from being subclassed.	Prevents subclasses from overriding the method.	The variable becomes a constant.
Purpose	Ensures the class's implementation cannot be altered by inheritance.	Ensures the method's behavior remains consistent in subclasses.	Ensures the variable's value remains unchanged.
Use Case Example	public final class String { ... }	public final void display() { ... }	final int MAX_SIZE = 100;
Compile-Time Effect	Throws a compile-time error if an attempt is made to extend the class.	Throws a compile-time error if a subclass tries to override the method.	Throws a compile-time error if code attempts to reassign the variable.
Common Usage	Used for utility classes like String, Math.	Used when defining behavior that must not change.	Used for constants or values that must remain fixed.

Let me know if you'd like further clarification on any of these!

final variable should be initialized(in the field or in the constructor)

How to create immutable class

To create an **immutable class** in Java, follow these steps:

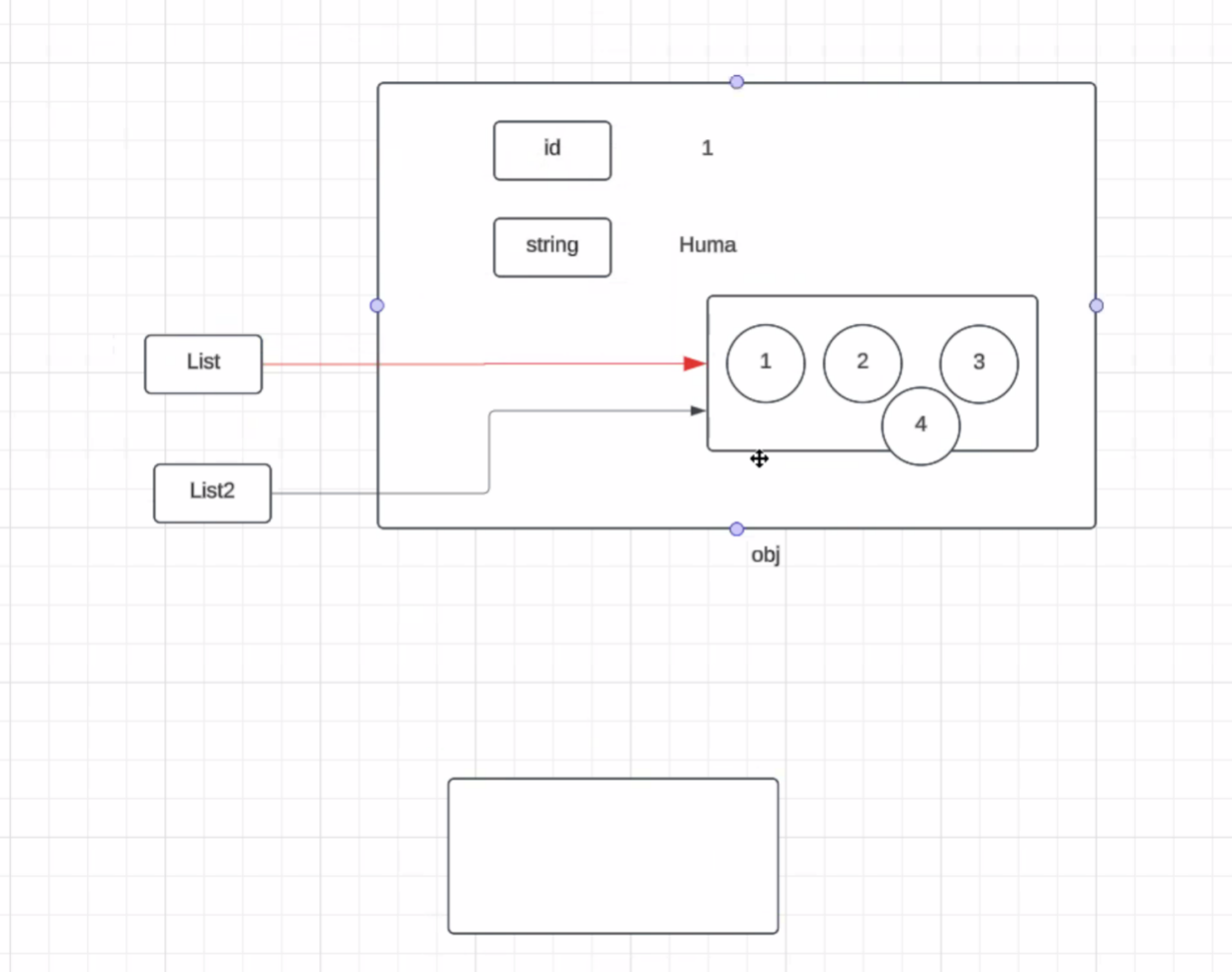
Steps to Create an Immutable Class

1. **Declare the class as final**:
    - **Prevents the class from being subclassed.**
      public final class ImmutableClass {
      }
2. **Declare all fields as private and final**:
    - **Ensures fields cannot be modified after initialization.**
      private final int id;
      private final String name;
3. **Initialize all fields through a constructor**:
    - **Set field values during object creation and do not allow modification later.**
      public ImmutableClass(int id, String name) {
      this.id = id;
      this.name = name;
      }
4. **Do not provide setter methods**:
    - **This ensures fields cannot be updated after object creation.**
5. **Provide only getter methods**:
    - **Return the field values, but avoid returning references to mutable objects.**
      public int getId() {
      return id;

```
}

public String getName() {
  return name;
}
```

6. **If the class contains mutable fields, make defensive copies**:
   - **Return a copy of the mutable object to prevent outside modification.**

```
private final List<String> list;

public ImmutableClass(int id, String name, List<String> list) {
  this.id = id;
  this.name = name;
  this.list = new ArrayList<>(list); // Defensive copy
}

public List<String> getList() {
  return new ArrayList<>(list); // Return a defensive copy
}
```



Example of an Immutable Class

```
import java.util.List;
import java.util.ArrayList;

public final class ImmutableClass {
   private final int id;
   private final String name;
   private final List<String> list; // Mutable field

public ImmutableClass(int id, String name, List<String> list) {
      this.id = id;
      this.name = name;
      this.list = new ArrayList<>(list); // Defensive copy
   }
```

```
public int getId() {
    return id;
  }

public String getName() {
    return name;
  }

public List<String> getList() {
    return new ArrayList<>(list); // Return a defensive copy
  }
}
```

Benefits of Immutable Classes

- **Thread-safety**: Immutable objects are inherently thread-safe.
- **Easier debugging**: State doesn't change, making it easier to reason about behavior.
- **Cache-friendly**: Objects can be cached and reused safely.

Let me know if you'd like more examples or a deeper explanation!

Final vs finally vs finalize

Here's a clear comparison of **final**, **finally**, and **finalize** in Java:

Feature	**final**	**finally**	**finalize**
Definition	A keyword used to declare constants, prevent inheritance, or method overriding.	A block in exception handling that executes regardless of exceptions.	A method in the Object class, used to clean up resources before an object is garbage collected.
Purpose	To restrict access, prevent changes, or define constants.	To ensure cleanup code executes (e.g., closing files, releasing locks).	Allows object cleanup before garbage collection.

Usage Context	Used with **classes**, **methods**, and **variables**.	Used with **try-catch** blocks.	Used with object lifecycle and garbage collection.
Behavior	- Final class: Cannot be subclassed.- Final method: Cannot be overridden.- Final variable: Cannot be reassigned.	Always executes whether an exception is thrown or not (except for system termination).	Automatically called by the garbage collector on an object before final removal.
Syntax	final int x = 10;final void method() {}final class MyClass {}	java try { ... } finally { ... }	java @Override protected void finalize() { ... }
Use Case Example	public final class String {}final int MAX = 100;	java try { ... } finally { close(); }	java protected void finalize() { cleanup(); }
Compile-Time Effect	Compilation error if: - Attempt to extend a final class.- Override a final method.- Reassign a final variable.	Ensures the finally block is executed after try or catch.	No direct compile-time effect but overrides need to use correct signature.
Common Pitfalls	Misuse of final can lead to immutability where not needed.	May not execute if the JVM is forcibly terminated.	Rarely used, as it is unpredictable and not a reliable cleanup mechanism.
Replacement Alternatives	- Immutable collections or patterns for constants.- Interface and composition for inheritance.	Use **try-with-resources** for resource management in Java 7+.	Use explicit cleanup methods (like close()) for resource management.

Key Points to Remember

1. **final**: A modifier used to restrict inheritance, overriding, or reassigning values.

2. **finally**: A block for guaranteed execution of cleanup code.
3. **finalize**: A method called by the garbage collector but is rarely used due to better alternatives like **try-with-resources** or explicit cleanup methods.

Would you like examples for each?

Static

Here's a comprehensive comparison of **static class**, **static method**, **static variable**, and **static block** in Java:

Feature	Static Class	Static Method	Static Variable	Static Block
Definition	A nested class declared as static.	A method declared with the static keyword.	A variable declared with the static keyword.	A block of code declared as static that executes when the class is loaded.
Scope	Belongs to the enclosing class and does not require an instance of the enclosing class.	Belongs to the class, not to any instance.	Shared across all instances of the class.	Executes once, at class loading time, for initialization purposes.
Purpose	To organize utility functions or constants as part of an enclosing class	To define behaviors that do not depend	To store data or constants shared across all objects of the class.	To initialize static variable

	without requiring its instance.	on instance data or state.		s or perform other setup operations when the class is loaded into memory.
Access Requirement	Accessed using the enclosing class name.	Invoked using the class name (or an instance reference, though not recommended).	Accessed using the class name (or an instance reference, though not recommended).	Automatically executed by the JVM during class loading. No explicit call required.
Initialization	Can be instantiated like any other class but doesn't require an enclosing class object.	Does not require an instance of the class.	Initialized once when the class is loaded into memory.	Executes at class loading time, before any method (including main)

				is called.
Key Limitations	Cannot access non-static members of the enclosing class directly.	Cannot use instance variables or methods unless explicitly passed.	May lead to unintended behavior if improperly synchronized in multithreaded environments.	Can only initialize static members; cannot contain non-static variables or methods.
Usage	For organizing code logically or grouping related helper methods and constants.	For operations independent of instance-specific data, such as utility or factory methods.	For defining constants, counters, configuration values, or shared state.	For initializing static fields or performing static setup logic when the class is loaded.
Execution	Instantiated like any normal class but without needing an instance of the outer class.	Invoked directly using the class name.	Automatically initialized during class loading.	Executed once when the class is loaded into the

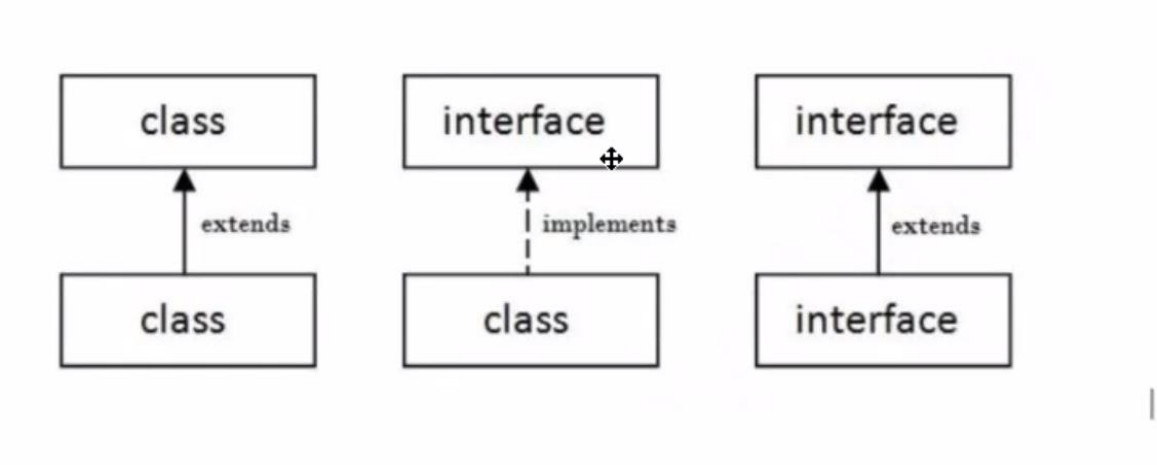
				JVM memory.
Example Syntax	java static class Nested { ... }	java static void method() { ... }	java static int count = 0;	java static { /* initialization logic */ }
Example Use Case	OuterClass.StaticClass.method();	ClassName.staticMethod();	System.out.println(ClassName.staticVariable);	Useful for initializing database connections, loading configuration, or setting up static constants.

Key Differences

- **Static Class**: Organizes utility methods or constants logically inside an outer class.
- **Static Method**: Defines operations that do not depend on instance-specific state.
- **Static Variable**: Stores shared data or constants accessible across all instances of a class.
- **Static Block**: Executes initialization code once, when the class is loaded into memory.

Would you like detailed code examples for any of these concepts?

Implements vs extends



In Java, both implements and extends are keywords used to establish relationships between classes and interfaces, but they are used in different contexts:

1. extends:

- Used when a class inherits from another class (class-to-class inheritance).
- A subclass inherits the properties (fields) and behaviors (methods) of the parent class.
- A class can only extend one other class because Java does not support multiple inheritance for classes.

Example:

```
class Animal {
  void eat() {
    System.out.println("Eating...");
  }
}

class Dog extends Animal {
  void bark() {
    System.out.println("Barking...");
  }
}
```

Here, the Dog class extends the Animal class and inherits its eat method.

2. implements:

- Used when a class agrees to fulfill the contract of an interface.
- A class can implement multiple interfaces, which allows Java to support a form of multiple inheritance at the interface level.
- A class that implements an interface must provide concrete implementations for all the methods declared in that interface (unless the class is abstract).

Example:

```
interface Animal {
  void eat();
}

class Dog implements Animal {
  public void eat() {
    System.out.println("Eating...");
  }

void bark() {
    System.out.println("Barking...");
  }
}
```

Here, the Dog class implements the Animal interface and provides the concrete implementation of the eat method.

Key Differences:

- extends: Used for inheriting from a **class** (single inheritance).
- implements: Used for fulfilling the contract of an **interface** (can implement multiple interfaces).

In summary, use extends for class inheritance and implements for implementing interfaces.

Oop

Object-Oriented Programming (OOP) in Java is a programming paradigm that organizes code into objects that represent real-world entities. Java is a fully object-oriented language, meaning it uses classes and objects as its fundamental building blocks. Here are the key principles of OOP in Java:

1. Classes and Objects

- **Class**: A blueprint for creating objects (instances). It defines properties (fields) and behaviors (methods).
- **Object**: An instance of a class. It represents a specific entity that can hold state and perform actions defined in its class.

Example:

```
class Dog {
  String name;  // Property (field)

  // Method (behavior)
  void bark() {
    System.out.println(name + " is barking!");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();  // Creating an object of the Dog class
    myDog.name = "Buddy";   // Setting the field
    myDog.bark();           // Calling the method
  }
}
```

2. Encapsulation

- Encapsulation refers to bundling the data (fields) and methods that operate on the data into a single unit, i.e., a class. It also involves restricting direct access to some of an object's components and allowing controlled access through methods (getters and setters).
- **Private fields** and **public methods** allow controlled access to the fields of a class.

Example:

```
class Person {
  private String name;  // Private field

// Getter and setter methods for name
  public String getName() {
    return name;
  }
```

```
public void setName(String name) {
    this.name = name;
  }
}
```

3. Inheritance

- Inheritance allows a new class (subclass) to inherit properties and methods from an existing class (superclass). This promotes code reuse.
- A subclass can also **override** methods of its superclass to provide a specific implementation.

Example:

```
class Animal {
  void eat() {
    System.out.println("Eating...");
  }
}

class Dog extends Animal {
  // Overriding the eat method
  @Override
  void eat() {
    System.out.println("Dog is eating...");
  }
}
```

4. Polymorphism

- Polymorphism allows one interface to be used for a general class of actions. The two types of polymorphism are:
    - **Method Overloading**: Defining multiple methods with the same name but different parameters in a class.(compile time)
    - **Method Overriding(run time)**: Redefining a method in the subclass that is already defined in the superclass.
- Polymorphism allows a subclass object to be treated as an object of its superclass, and the correct method is called at runtime.

Example:

```
class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}

class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Dog barks");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();
    Animal myDog = new Dog(); // Polymorphism: Dog is treated as Animal

    myAnimal.sound();  // Animal makes a sound
    myDog.sound();     // Dog barks
  }
}
```

5. Abstraction

- Abstraction hides complex implementation details and shows only essential features. This can be achieved using:
    - **Abstract classes**: A class that cannot be instantiated directly and may contain abstract methods (methods without a body) that subclasses must implement.
    - **Interfaces**: A contract that defines methods without implementation, and classes that implement the interface must provide implementations for those methods.

Example of Abstract Class:

```
abstract class Animal {
  abstract void sound();  // Abstract method
}
```

```
class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Dog barks");
  }
}
```

Example of Interface:

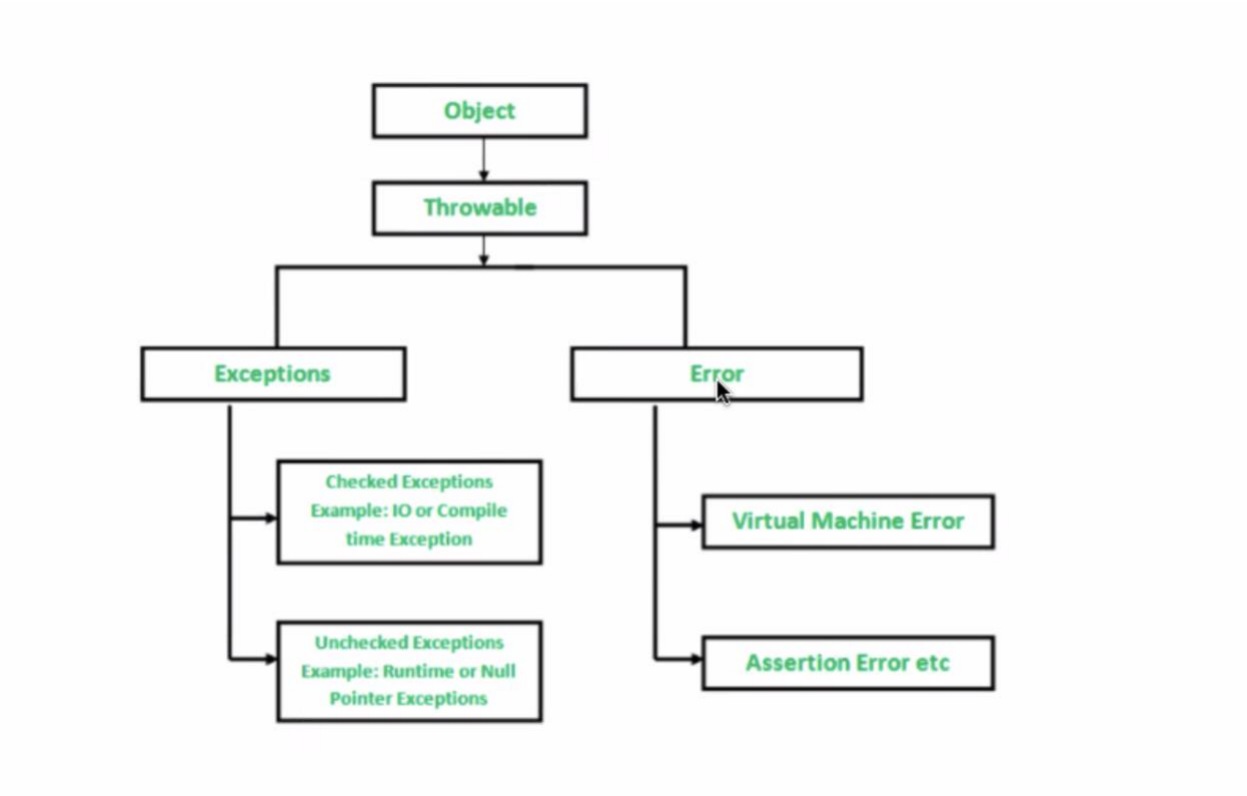
```
interface Animal {
  void sound();  // Interface method
}

class Dog implements Animal {
  @Override
  public void sound() {
    System.out.println("Dog barks");
  }
}
```

Key Concepts in OOP:

- **Class**: A blueprint for creating objects, containing fields and methods.
- **Object**: An instance of a class.
- **Encapsulation**: The bundling of data and methods that operate on the data into a single unit, with access control.
- **Inheritance**: The ability for a class to inherit methods and properties from another class.
- **Polymorphism**: The ability for different classes to be treated as instances of the same class through method overriding or overloading.
- **Abstraction**: Hiding the implementation details and showing only the necessary features.

In summary, OOP in Java revolves around organizing code into classes and objects and following the principles of encapsulation, inheritance, polymorphism, and abstraction. This approach helps to build modular, reusable, and maintainable code.

```
OOP
    - Inheritance: implements, extends
    - Encapsulation: private, getter/ setter
    - Abstraction: abstract class, interface
    - Polymorphism: override, overload
```



Don' t handle error, just let it crash

But we will handle exception

Checked/unchecked exception

In Java, exceptions are divided into two main categories: **checked exceptions** and **unchecked exceptions**. Understanding the difference is key to handling exceptions effectively in Java programs.

1. Checked Exceptions

- **Definition**: Checked exceptions are exceptions that are checked at compile-time. The compiler requires that you either handle (catch) the exception using a try-catch block or declare it in the method signature using the throws keyword.
- **Usage**: These exceptions typically represent recoverable conditions that the programmer should anticipate and handle, such as invalid user input, file not found, or network errors.
- **Examples**: IOException, SQLException, ClassNotFoundException, FileNotFoundException, ParseException.

Example of Checked Exception:

```
import java.io.*;

public class CheckedExceptionExample {
   public static void main(String[] args) {
      try {
         FileReader file = new FileReader("somefile.txt"); // may throw FileNotFoundException
         BufferedReader fileInput = new BufferedReader(file);
         System.out.println(fileInput.readLine());
         fileInput.close();
      } catch (IOException e) {  // Handling the checked exception
         System.out.println("An error occurred: " + e.getMessage());
      }
   }
}
```

In the above example, the FileNotFoundException is a checked exception, and it must be either caught in a try-catch block or declared to be thrown in the method signature using throws.

2. Unchecked Exceptions (Runtime Exceptions)

- **Definition**: Unchecked exceptions, also known as **runtime exceptions**, are not checked at compile-time. These exceptions occur at runtime and are typically the result of programming errors, such as invalid array accesses, null pointer dereferences, or illegal argument values.
- **Usage**: These exceptions usually represent programming bugs that should be fixed by the developer (e.g., NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException).

- **Examples**: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, IllegalArgumentException, ClassCastException.

Example of Unchecked Exception:

```
public class UncheckedExceptionExample {
  public static void main(String[] args) {
    try {
      int[] numbers = new int[3];
      numbers[5] = 10; // May throw ArrayIndexOutOfBoundsException
    } catch (Exception e) {
      System.out.println("An error occurred: " + e.getMessage());
    }
  }
}
```

In this example, ArrayIndexOutOfBoundsException is an unchecked exception. Although it's caught here, this type of exception is not required to be explicitly handled or declared.

Key Differences Between Checked and Unchecked Exceptions:

Aspect	**Checked Exceptions**	**Unchecked Exceptions**
Definition	Exceptions that are checked at compile-time.	Exceptions that occur at runtime.
Subclass of	Exception but not RuntimeException.	RuntimeException or its subclasses.
Handling	Must be explicitly handled with try-catch or declared with throws.	Can be handled, but not required.
Examples	IOException, SQLException, ClassNotFoundException.	NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.
Common Causes	External conditions (e.g., file I/O, database access).	Programming errors (e.g., invalid array indices, null pointer dereference).

Example	FileNotFoundException, IOException	NullPointerException, IllegalArgumentException

Summary:

- **Checked Exceptions** are recoverable and must be explicitly handled or declared to be thrown.
- **Unchecked Exceptions** are typically caused by programming errors, and you are not required to explicitly handle or declare them.

How to handle exception

In Java, exceptions can be handled using **try-catch blocks**. When an exception occurs, the program flow is interrupted, and the exception can be caught and processed in a catch block. You can also use a finally block to execute code that should run regardless of whether an exception occurs or not.

Here's an overview of how to handle exceptions in Java:

1. Try-Catch Block

- The try block contains the code that might throw an exception.
- The catch block handles the exception if it occurs.

Syntax:

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
}
```

Example:

```
public class ExceptionHandlingExample {
   public static void main(String[] args) {
      try {
         int result = 10 / 0;  // This will throw ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Error: Cannot divide by zero!");
      }
```

```
    }
}
```

Output:

Error: Cannot divide by zero!

2. Multiple Catch Blocks

- You can handle multiple exceptions by using multiple catch blocks.
- The catch blocks are evaluated in the order they appear, and the first one that matches the exception is executed.

Example:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length());  // This will throw NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Null pointer exception occurred!");
        } catch (Exception e) {
            System.out.println("Some other exception occurred!");
        }
    }
}
```

Output:

Null pointer exception occurred!

3. Catch Multiple Exceptions in One Block (Java 7 and later)

- Starting from Java 7, you can handle multiple exceptions in a single catch block using the pipe | symbol.

Example:

```
public class MultiExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            String str = null;
            int result = 10 / 0;  // Both NullPointerException and ArithmeticException may occur
```

```
    } catch (NullPointerException | ArithmeticException e) {
      System.out.println("Error: " + e.getClass().getSimpleName());
    }
  }
}
```

Output:

Error: ArithmeticException

4. Finally Block

- The finally block is used to execute code that must run regardless of whether an exception is thrown or not. It's typically used for cleanup operations, such as closing resources (files, database connections, etc.).
- Even if an exception is thrown or caught, the finally block will always execute.

Syntax:

```
try {
  // Code that may throw an exception
} catch (ExceptionType e) {
  // Handle the exception
} finally {
  // Code that will always execute
}
```

Example:

```
public class FinallyExample {
  public static void main(String[] args) {
    try {
      System.out.println("Trying...");
      int result = 10 / 0;  // This will throw ArithmeticException
    } catch (ArithmeticException e) {
      System.out.println("Error occurred: " + e.getMessage());
    } finally {
      System.out.println("This will always execute.");
    }
  }
}
```

Output:

Trying...
Error occurred: / by zero
This will always execute.

5. Throwing Exceptions

- You can explicitly throw exceptions in Java using the throw keyword.
- When you throw an exception, it must either be caught by a catch block or declared in the method signature using the throws keyword.

Example:

```
public class ThrowExample {
  public static void main(String[] args) {
    try {
      validateAge(15);  // Throws an exception
    } catch (IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
  }



  public static void validateAge(int age) {
    if (age < 18) {
      throw new IllegalArgumentException("Age must be 18 or older.");
    }
    System.out.println("Valid age.");
  }
}
```

Output:

Age must be 18 or older.

6. Throws Keyword

- The throws keyword is used to declare that a method can throw exceptions, so the caller of the method is aware of the potential exception and can handle it.

- You use throws in the method signature to specify exceptions that can be thrown but are not necessarily handled inside the method.

Example:

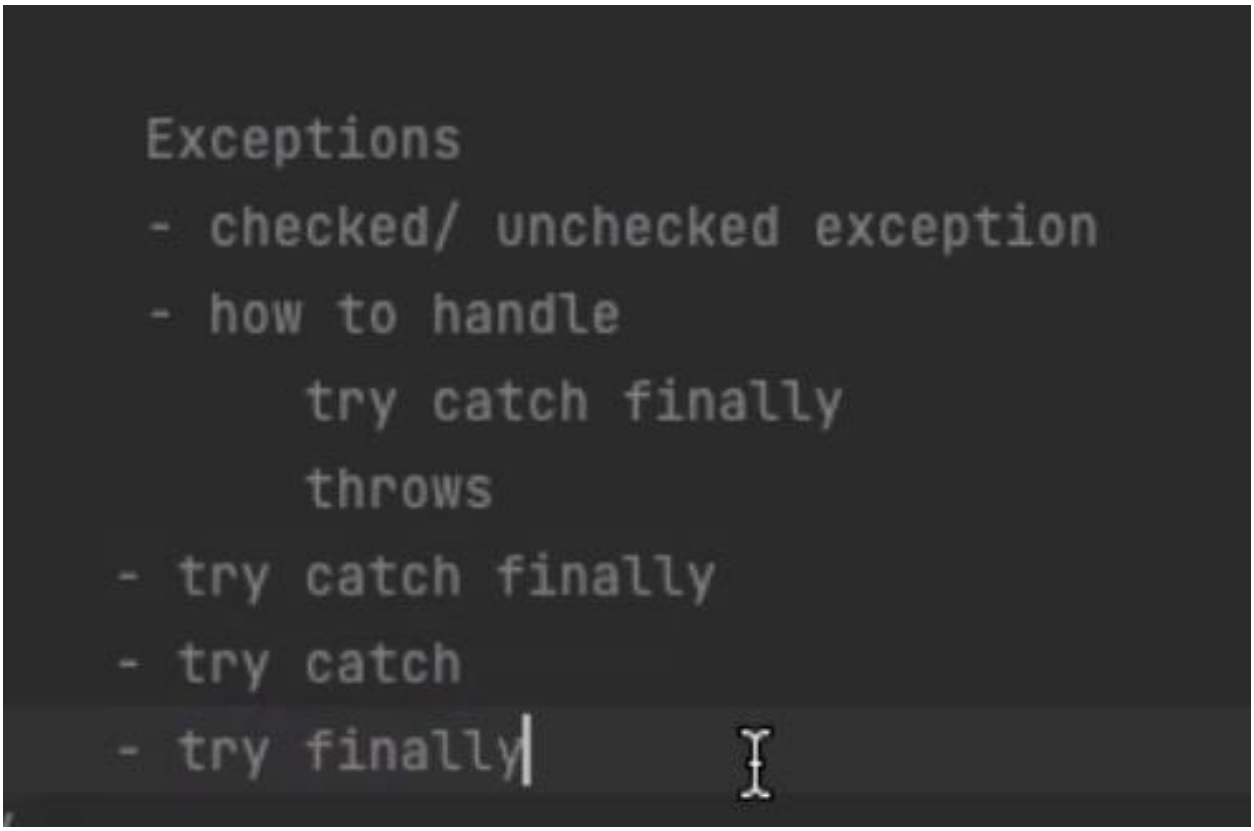
```
public class ThrowsExample {
  public static void main(String[] args) {
    try {
      readFile("nonexistentfile.txt");
    } catch (IOException e) {
      System.out.println("File not found!");
    }
  }

public static void readFile(String filename) throws IOException {
    FileReader file = new FileReader(filename);  // Might throw FileNotFoundException
    BufferedReader reader = new BufferedReader(file);
    reader.readLine();
    reader.close();
  }
}
```

In the above example, readFile declares that it might throw an IOException, which is handled by the calling method using try-catch.

Summary of Exception Handling:

- **try-catch**: Used to handle exceptions. The catch block catches specific exceptions and provides code to handle them.
- **finally**: Always executed, regardless of whether an exception occurs or not. Typically used for cleanup code.
- **throw**: Used to explicitly throw an exception.
- **throws**: Used in method declarations to specify that a method can throw exceptions.

By properly handling exceptions, you can make your Java program more robust and prevent it from crashing unexpectedly due to runtime issues.


Exceptions
- checked/ unchecked exception
- how to handle
try catch finally
throws
- try catch finally
- try catch
- try finally