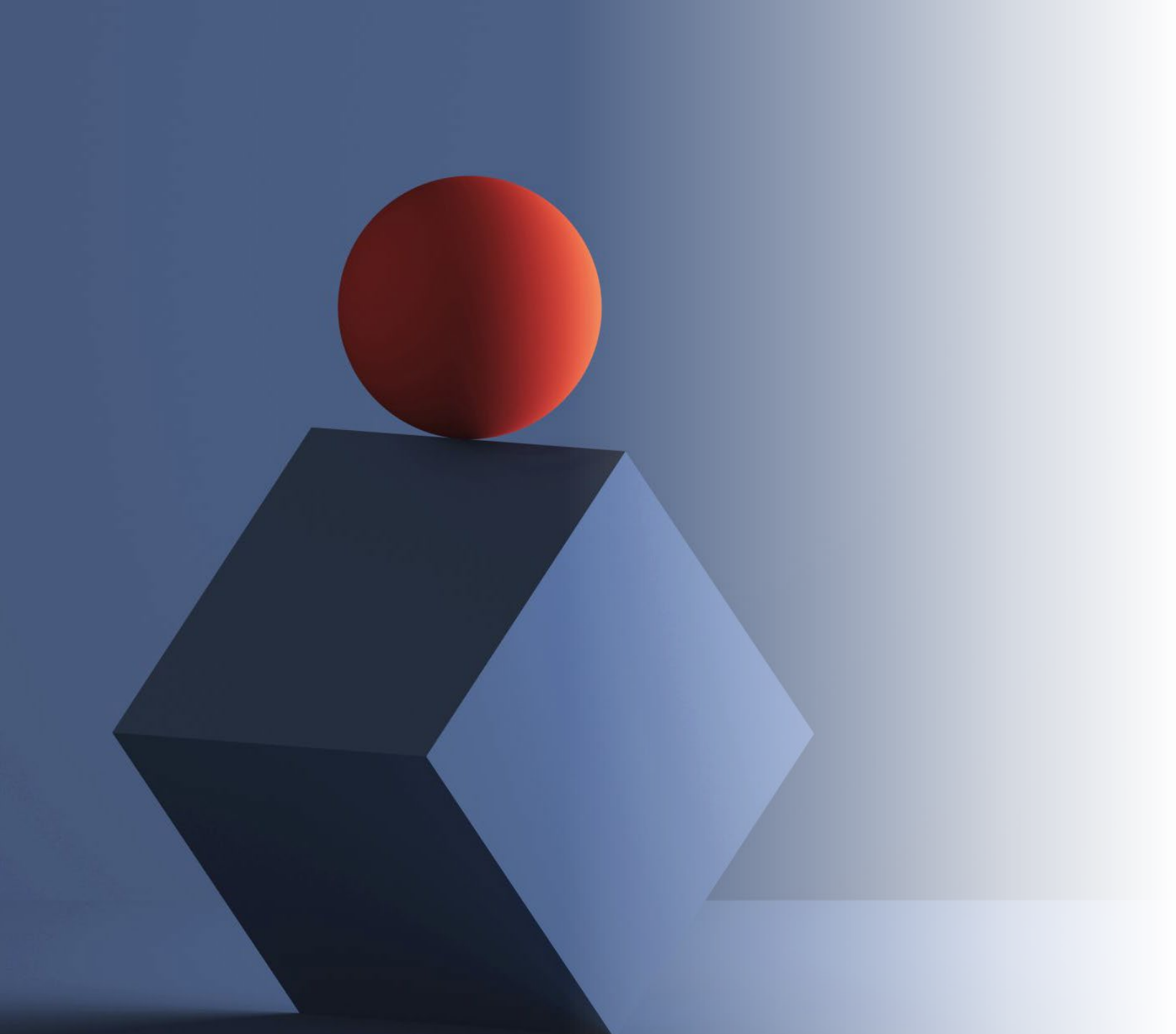
# Summary of knowledge prior Final

University American College Skopje
School of Computer Science and Information Technology
Course: Programming Languages
Prepared by: Dejan Mitov



# Polymorphism, Interfaces

# Understanding Polymorphism in Java

- **Definition:** Polymorphism allows methods to perform differently based on the object it interacts with, enhancing code flexibility and reusability.
- **Core Concept:** It's a fundamental aspect of object-oriented programming, enabling objects of different classes to respond differently to the same method call.
- **Real-World Analogy:** Like a person speaking different languages based on the country they are in, a polymorphic method adapts its behavior based on the object it's applied to.

# Types of Polymorphism in Java

- **Static Polymorphism (Compile-time):** Achieved through method overloading where methods in the same class have the same name but different parameters.
- **Dynamic Polymorphism (Run-time):** Achieved through method overriding, where a subclass provides a specific implementation for a method already defined in its superclass.

# Polymorphism Through Inheritance

- Inheritance & Polymorphism: Demonstrates how superclass references can point to subclass objects, allowing for dynamic method invocation.
- Example: `Base` class with a method `print()` is extended by `Derived` class, which overrides `print()`. When calling `print()` on a `Base` reference pointing to a `Derived` object, the `Derived` version is executed.

# Exploring Method Overriding

- **Concept:** Subclass method overrides a superclass method with the same signature.
- **Rules:** Method must have the same name, return type, and parameters. Access level can't be more restrictive.

# Dynamic (Late) Binding

- **Definition:** Java runtime determines the method to execute based on the object's actual class type, not the reference type.
- **Mechanism:** Enables polymorphism by binding method calls to appropriate method definitions at runtime, not compile time.

# Java Interfaces - A Gateway to Multiple Inheritance

- Role of Interfaces: Provide a contract for classes to implement. They contain abstract methods that must be overridden by implementing classes.
- Multiple Inheritance: Java interfaces facilitate a form of multiple inheritance, allowing a class to implement multiple interfaces.
- Characteristics: Interfaces contain only abstract methods, and all members are implicitly public.

# Implementing Java Interfaces

- How to Implement: Classes use the `implements` keyword to adhere to the interface's contract.
- Obligations: Implementing class must provide concrete implementations for all abstract methods declared in the interface.

# Using Interfaces as Data Types

- **Concept:** Interfaces can be used as data types to create more flexible and loosely coupled code.
- **Polymorphism with Interfaces:** Allows different classes to be treated uniformly, as long as they implement the same interface.

# Leveraging Interfaces in Method Arguments

- **Method Arguments:** Interfaces used as method parameters increase the flexibility and reusability of the method.
- **Advantages:** Enables methods to accept a wider range of objects, as long as they conform to the interface.

# Polymorphism and Interfaces - Best Practices

- **Design with Interface-Based Polymorphism:** Favor interface-based design to abstract away implementation details, allowing for more flexible and testable code.
- **Use Polymorphism Carefully:** Understand when to use overloading vs. overriding.
- **Implement Interfaces Consistently:** When a class implements an interface, ensure it adheres to the contract fully and clearly. Avoid partially implemented interfaces unless the class is abstract.
- **Document Behavior:** Clearly document the behavior of methods, especially when overriding, to inform other developers of your implementation's nuances.

# Reflection. IoC

# Understanding Java Reflection

- **Definition:** Reflection is the ability of a program to analyze itself, particularly its classes, interfaces, and methods during runtime.
- **Use Cases:** It's crucial for advanced Java applications, enabling dynamic behavior and analysis of program structure.
- **Code Example:** Class<?> c = Class.forName("java.util.ArrayList");

# Core Capabilities of Java Reflection

- **Runtime Examination:** Inspecting classes, fields, methods, and constructors at runtime.
- **Dynamic Execution & Access:** Modifying the runtime behavior of a class, including instantiating objects, accessing fields, and invoking methods dynamically.
- **Code Example:** Method method = c.getDeclaredMethod("size");

# Accessing Class Information via Reflection

- **Obtaining Class Instances:** Using `getClass()`, `Class.forName()`, and `.class` literals.
- **Inspecting Class Details:** Exploring a class's methods, fields, constructors, and superclass.
- **Code Example:**
  - Class<?> arrayListClass = ArrayList.class;
    Method[] methods = arrayListClass.getDeclaredMethods()

# Manipulating Fields and Methods Using Reflection

- **Field Access:** Inspecting and modifying object fields regardless of their usual access restrictions.
- **Method Invocation:** Dynamically invoking methods of a class.
- **Code Example:**
  - Field field = c.getDeclaredField("size");
  - field.setAccessible(true);

# Dynamic Object Creation with Reflection

- **Instantiating Classes Dynamically:** Using `Class.newInstance()` or constructors.
- **Use Cases:** Essential for frameworks, tools, and libraries that require flexibility.
- **Code Example:**
  - Constructor<?> constructor = c.getConstructor();
  - Object instance = constructor.newInstance();

# Dynamic Type Loading in Java

- **Definition:** Loading classes during runtime rather than at compile-time.
- **Benefits:** Enhances modularity, reduces dependencies, and allows for more flexible application structures.
- **Code Example:**
  - Class<?> dynamicClass = Class.forName("com.example.MyClass");

# Implementing Inversion of Control (IoC)

- **Concept of IoC:** Decoupling the execution of a task from its implementation.
- **Role in Frameworks:** Foundations of many Java frameworks like Spring and Hibernate.

# IoC Containers and Dependency Injection

- **IoC Containers:** Manages the instantiation and lifecycle of objects (beans in Spring).
- **Dependency Injection:** Automatically injecting dependencies into objects managed by the IoC container.

# Role of Reflection in Dynamic Type Loading and IoC

- **Dynamic Behavior:** Reflection enables dynamic type loading and IoC patterns by allowing runtime analysis and modification.
- **Code Example:**
  - Class<?> workerClass = Class.forName("com.example.MyWorker");
  - Worker workerInstance = (Worker) workerClass.newInstance();

# Summary: Reflection, Dynamic Type Loading, and IoC in Java

- Reflection: A powerful feature enabling Java programs to inspect and manipulate their own classes and objects at runtime.
- Dynamic Type Loading: This technique allows Java applications to load classes dynamically at runtime, enhancing flexibility and modularity.
- Inversion of Control (IoC): A design principle where control flow is inverted, meaning custom-written portions of a program receive the flow of control from a generic framework.


# Java Serialization

# Understanding Java Serialization

- **Definition:** Serialization is the process of converting an object into a byte stream, enabling it to be easily saved to a file, database, or transferred over a network.
- **Usage:** Essential for persisting object states, caching, or sending objects between different JVMs.
- **Code Example:** `ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("object.dat"));`

# Implementing the Serializable Interface

- **Basics:** Serializable is a marker interface; it doesn't contain any methods.
- **Purpose:** Informs the Java serialization mechanism that a class is serializable.
- **Code Example:**
  - public class MyObject implements Serializable {
  - private int data;
  - // getters and setters
  - }

# The Process of Serializing an Object

- **Steps:** How to serialize an object using `ObjectOutputStream`.
- **Considerations:** Handling `IOException` and ensuring object's fields are serializable.
- **Code Example:**
  - MyObject obj = new MyObject();
  - oos.writeObject(obj);
  - oos.close();

# Deserializing - Bringing Objects Back to Life

- **Process:** Using `ObjectInputStream` to read and convert byte streams back to objects.
- **Important Notes:** Ensuring class compatibility and handling `ClassNotFoundException`.
- Code Example:
  - ObjectInputStream ois = new ObjectInputStream(new FileInputStream("object.dat"));
  - MyObject obj = (MyObject) ois.readObject();
  - ois.close();

# serialVersionUID and Version Control

- **Role:** Ensures that a deserialized object is compatible with the current class version.
- **Best Practice:** Declaring a `serialVersionUID` for every serializable class.
- **Code Example:**
  - private static final long serialVersionUID = 1L;

# Implementing Custom Serialization

- When to Use: Customizing serialization logic for complex scenarios.
- Methods: Implementing `writeObject` and `readObject`.
- Code Example:
  - private void writeObject(ObjectOutputStream oos) throws IOException {
  - // custom write logic
  - }
  - private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
  - // custom read logic
  - }

# Managing Transient Fields in Serialization

- **Transient Keyword:** Marking fields that should not be serialized.
- **Usage:** For sensitive data or fields that don't make sense to persist.
- **Code Example:**
  - private transient String sensitiveData;

# Serialization in the Context of Inheritance

- **Inheritance:** Serializing objects from classes that are part of an inheritance hierarchy.
- **Superclass Serialization:** Handling when a superclass is not serializable.
- Code Example:
  - public class Parent { /* Non-serializable */ }
  - public class Child extends Parent implements Serializable { /* ... */ }

# Pitfalls in Serialization and How to Overcome Them

- **Common Issues:** Versioning problems, performance overhead, security concerns.
- **Solutions:** Using `serialVersionUID`, externalization, using `transient` fields effectively.

# Recap: Key Concepts of Serialization

- **Essence of Serialization:** Transforming Java objects into a byte stream to save state, transfer over networks, or store in databases.
- **Serializable Interface:** A marker interface used to indicate that a class's objects are eligible for serialization.
- **Common Processes:** Involves using ObjectOutputStream and ObjectInputStream for serializing and deserializing objects, respectively.
- **Handling serialVersionUID:** Ensures version compatibility during the deserialization process.

# Functional Programming

# Embracing Functional Programming in Java

- **Definition:** Functional programming (FP) is a programming paradigm that treats computation as the evaluation of mathematical functions.
- **Key Characteristics:** Focuses on expressions and declarations rather than execution of statements.
- **Java and FP:** Introduction of FP concepts in Java 8 with Lambda expressions and Stream API.

# Core Concepts in Functional Programming

- **Immutability:** Unchanging nature of data after its creation.
- **Pure Functions:** Functions where the return value is only determined by its input values, without observable side effects.
- **Higher-Order Functions:** Functions that take other functions as parameters or return them as results.

# Lambda Expressions: A Pillar of FP in Java

- **Definition:** Anonymous functions that provide a concise and flexible way to implement function interfaces.
- **Syntax and Usage:** `(parameters) -> expression` or `(parameters) -> { statements; }`
- Code Example:
  - (int a, int b) -> a + b

# Stream API: Harnessing Functional Power

- **Overview:** A key API in Java that supports functional-style operations on streams of elements.
- **Key Operations:** `filter`, `map`, `collect`.
- **Code Example:**
- List<String> collected = list.stream()
- .map(String::toUpperCase)
- .collect(Collectors.toList());

# The Role of Immutability in FP

- **Importance of Immutability:** Ensuring reliability and predictability of code.
- **Achieving Immutability:** Creating classes where state cannot be modified after instantiation.
- **Code Example:**
- final class ImmutableValue {
- private final int value;
- public ImmutableValue(int value) { this.value = value; }
- }

# Pure Functions and Side-Effect Free Code

- **Defining Pure Functions:** Functions that do not cause any observable side effects.
- **Benefits:** Easier to test, debug, and parallelize.
- **Code Example:**
- int add(int a, int b) { return a + b; }

# Exploring Higher-Order Functions

- **Concept:** Functions that operate on other functions by taking them as arguments or returning them.
- **Practical Use:** Widely used in operations like `map`, `filter`.
- Code Example:
- public <T, R> List<R> map(List<T> list, Function<T, R> func) {
-  List<R> result = new ArrayList<>();
-  for (T item : list) {
-  result.add(func.apply(item));
-  }
-  return result;
-  }

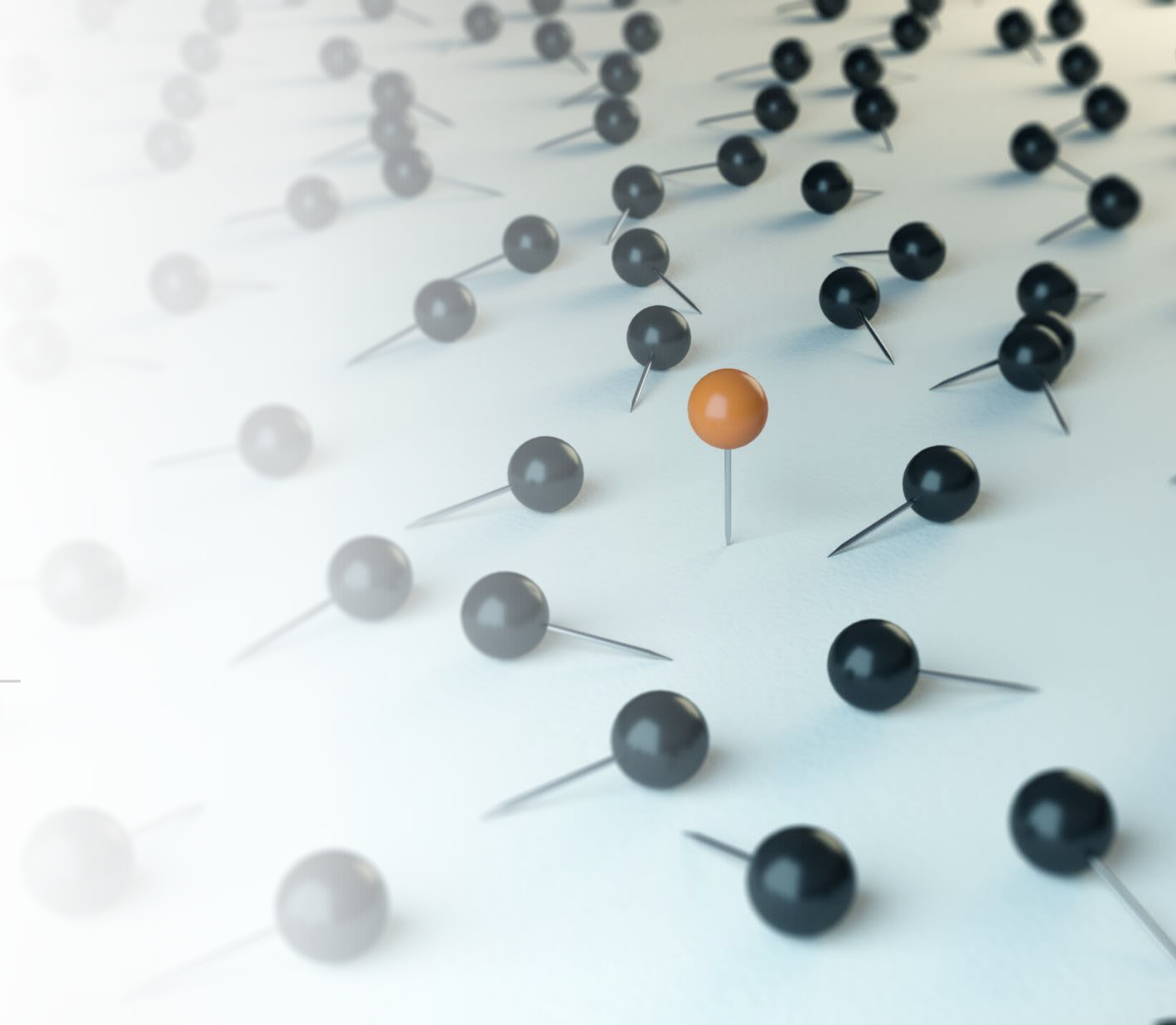
# Best Practices in Functional Programming

- **Stateless Behavior:** Encouraging the use of stateless lambda expressions and functions.
- **Readability and Simplicity:** Writing clear and concise functional code.

# The Future of Functional Programming in Java

- **Java's Evolution in FP:** Each new Java version introduces enhancements in functional programming (FP), making FP more integral and efficient.
- **Recent Developments:** Features like Streams API, Optional class, and more expressive lambda expressions.
- **Java's Roadmap:** Anticipate future Java releases to further refine and expand FP capabilities, incorporating feedback from the developer community and emerging software trends.

# Exceptions



# Understanding Exceptions in Java

- **Definition:** An exception is an event that disrupts the normal flow of a program's instructions.
- **Importance:** Handling exceptions is crucial for robust and error-resistant software.

# Using try-catch for Exception Handling

- **Basic Concept:** Encapsulating code that may throw an exception within a `try` block, and handling it in a `catch` block.
- **Multiple catch Blocks:** Handling different types of exceptions separately.
- **Code Example:**
- try {
- // code that may throw an exception
- } catch (ExceptionType1 e) {
- // handling code
- } catch (ExceptionType2 e) {
- // different handling
- }

# Understanding the finally Block

- Purpose: Ensuring a block of code is executed after a try-catch block, regardless of whether an exception is thrown.
- Use Cases: Commonly used for resource cleanup, like closing file streams.
- Code Example:
- try {
-  // risky code
-  } catch (Exception e) {
-  // exception handling
-  } finally {
-  // cleanup code, always executed
-  }

# Defining Custom Exceptions

- **Need for Custom Exceptions:** Tailoring exceptions to specific application needs.
- **How to Create:** Extending the `Exception` or `RuntimeException` class.
- **Code Example:**
- public class MyCustomException extends Exception {
- public MyCustomException(String message) {
- super(message);
- }
- }

# Declaring Exceptions with throws

- **Usage:** Indicating that a method may throw an exception.
- **Effect on Callers:** Forcing calling methods to handle or further declare the exception.
- **Code Example:**
- public void riskyMethod() throws IOException {
- // code that may throw IOException
- }

# Exception Chaining in Java

- **Concept:** Wrapping one exception within another.
- **Benefits:** Preserving original exception information while throwing a new exception.
- **Code Example:**
- try {
- // code that throws SQLException
- } catch (SQLException e) {
- throw new MyCustomException("Database error", e);
- }

# Managing Resources with try-with-resources

- Introduction: Automatic resource management introduced in Java 7.
- Usage: Ensuring that resources are closed after execution, regardless of whether an exception occurs.
- Code Example:
- try (FileInputStream fis = new FileInputStream("file.txt")) {
- 	// use fis
- }

# Open discussion

# Polymorphism in Java

Discuss the concept of polymorphism in Java and its significance in object-oriented programming. Provide examples of static and dynamic polymorphism.

# Polymorphism Through Inheritance

Explain how polymorphism is achieved through inheritance, using the `Base` and `Derived` class example from the presentation.

# Method Overriding

Describe method overriding in Java. How does it differ from method overloading, and what are its rules and implications?

# Dynamic Binding

Explain the concept of dynamic (late) binding in Java and its role in enabling polymorphism

# Java Interfaces

Discuss the role of interfaces in Java and how they facilitate multiple inheritance. What are the key characteristics of interfaces?

# Implementing Interfaces

Describe the process and obligations involved in implementing interfaces in Java. Why is it important to implement interfaces consistently?

# Interfaces as Method Arguments

How do interfaces enhance method flexibility and reusability when used as method arguments? Provide examples.

# Reflection in Java

Define Java reflection and discuss its use cases. How does reflection enable dynamic behavior and structural analysis of programs?

# Dynamic Type Loading

Explain dynamic type loading in Java, its benefits, and provide an example scenario where it can be effectively utilized.

# Inversion of Control (IoC)

Describe the concept of Inversion of Control in Java

# Java Serialization

Explain the process of serialization in Java. What is the purpose of the `Serializable` interface and `serialVersionUID`?

# Custom Serialization

When is custom serialization needed in Java, and how is it implemented? Discuss the concept of transient fields in this context.

# Functional Programming in Java

Define functional programming in Java and discuss its key characteristics, such as immutability, pure functions, and higher-order functions.

# Lambda Expressions and Stream API

How have lambda expressions and the Stream API enhanced functional programming capabilities in Java? Provide examples to illustrate their usage.