2025

Understanding of Java and its Features

Topics encourage critical thinking and a deeper understanding of Java and its features, giving opportunity to see how they apply in practice

VLADYSLAV BOGDANTSEV

Contents

1) Type Safety:

1. What are the benefits and trade-offs of using generics in Java? When might it be appropriate to use raw types?
2. How does type erasure impact generic types in Java, and what are the implications for runtime type-checking?

1. Benefits and Trade-offs of Using Generics

Benefits:

- Type Safety: Generics provide stronger type checks at compile time, helping to catch type-related errors before runtime. This reduces the risk of ClassCastException in code.
- Code Reusability: Generics promote code reusability by allowing a single method or class to operate on different types, enhancing flexibility.
- Cleaner Code: Using generics can lead to cleaner and more readable code, as it eliminates the need for explicit casting.

Trade-offs:

- Complexity: Generics introduce a level of complexity in understanding type parameters, especially for developers new to the concept.
- Performance Considerations: While generics can improve type safety, they may introduce slight overhead due to type checks at compile-time and runtime. However, this is usually negligible.
- Type Erasure: Generics are implemented via type erasure, which means that generic type information is removed during compilation. This can lead to limitations when you need to know the actual type at runtime.

Raw Types:

- Raw types can be used for legacy code or when you have a situation where generics would complicate your code unnecessarily.
- Raw types can lead to runtime type-safety issues — the compiler will allow something that may not be safe.

2. Type Erasure Impact on Generic Types

Type Erasure:

- In Java, generic types are not reified; instead, they are replaced with their bounds (or Object if none are specified) at runtime. This means that the generic type information is not available at runtime.

Implications:

- No Runtime Type Information: Since type parameters are erased, you cannot create an instance of a type parameter, check its type with instanceof, or create dynamic arrays of a generic type.
- Generic Arrays: You cannot create arrays of a generic type (e.g., new T[10] is illegal). You would need to use collections instead.
- Type Safety: While generics enhance type safety at compile time, their absence at runtime means that wrong assumptions can lead to ClassCastException if types are not checked appropriately.

```java
/**
 * Raw Type Warning: The rawList demonstrates the potential pitfalls of using raw types.
 * It allows adding both String and Integer, but casting to String later can lead to a
ClassCastException.
 *
 * Type Safety Advantage: Using generics helps avoid such issues and ensures type safety at
compile time.
 * */
public class GenericsExample {

    private static final Logger log =
Logger.getLogger(String.valueOf(GenericsExample.class));

    //generic method to display list
    public static <T> void displayList(List<T> list) {
        for (T t : list) {
            System.out.println(t);
        }
    }

    //will cause a runtime issue if items are not of the expected type
    public static void displayRawList(List list) {
        try {
            for (Object object : list) {
                String s = (String) object;
                System.out.println(s);
            }
        } catch (ClassCastException e) {
            log.severe("ClassCastException: Object refer into TypeClass invalid");
        }
    }

    public static void main(String[] args) {
        //generic list
        List<String> stringList = new ArrayList<>();
        stringList.add("Hello");
        stringList.add("Generics");
        displayList(stringList);

        //raw list (not recommended)
        List rawList = new ArrayList(); // raw type
        rawList.add("String");
        rawList.add(10); // This compiles, but it can lead to issues
        displayRawList(rawList); // Unsafe cast can cause ClassCastException at runtime
    }
}
```

2) Polymorphism:

1. In what scenarios would you prefer method overloading over method overriding, or vice versa?
2. How does polymorphism enhance maintainability and scalability in object-oriented design?

1. Method Overloading vs. Method Overriding

Method Overloading:

- Overloading occurs when multiple methods have the same name but different parameter lists (different types or different numbers of parameters).
- It is a compile-time polymorphism, as the method to be executed is determined during compilation based on the method signature.

Use Cases:

- When you want to perform similar operations with slightly different data types or different numbers of arguments.
- For example, a print method that can print a string, an integer, or an array.

Method Overriding:

- Overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.
- It enables runtime polymorphism and is essential for achieving dynamic method dispatch in Java; the method invoked is determined at runtime based on the object's type.

Use Cases:

- When you want to provide a specific implementation of a method that is common to all subclasses.
- For example, an Animal class with an overridden makeSound method in subclasses like Dog and Cat.

2. Enhancing Maintainability and Scalability

Maintainability:

- Polymorphism allows you to write more generic and reusable code. By programming to an interface or superclass type, you can easily change the implementation without affecting the rest of your codebase. This leads to fewer code changes and simpler testing.

Scalability:

- Polymorphism facilitates the addition of new subclasses with minimal disruption to existing code. For example, adding a new Bird class that inherits from Animal doesn’t require changes to the code that uses the Animal type, as it can handle any subclass seamlessly.
- This makes it easier to extend functionality by adding additional classes and methods that conform to established behavior, which can greatly benefit larger systems and applications.

```java
// Abstract superclass
abstract class Shape {
    // Abstract method for area calculation
    abstract double area();
    // Method Overloading: Calculate area with different parameters
    double area(double radius) {
        return Math.PI * radius * radius; // Area of a Circle
    }
    double area(double length, double width) {
        return length * width; // Area of a Rectangle
    }
    double area(double base, double height, boolean isTriangle) {
        if (isTriangle) {
            return 0.5 * base * height; // Area of a Triangle
        }
        return 0; // Default case
    }
}
// Subclass for Circle
class Circle extends Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }
    @Override
    double area() {
        return area(radius); // Call overloaded method
    }
}
// Subclass for Rectangle
class Rectangle extends Shape {
    private double length;
    private double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    @Override
    double area() {
        return area(length, width); // Call overloaded method
    }
}
// Subclass for Triangle
class Triangle extends Shape {
    private double base;
    private double height;

    Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }
    @Override
    double area() {
        return area(base, height, true); // Call overloaded method
    }
}

public class PolymorphismShapes {
    public static void main(String[] args) {
        // Create shape objects
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);
        Shape triangle = new Triangle(3, 7);

        // Outputs the area of each shape
        System.out.println("Circle Area: " + circle.area()); // Output: Circle Area: 78.53981633974483
        System.out.println("Rectangle Area: " + rectangle.area()); // Output: Rectangle Area: 24.0
        System.out.println("Triangle Area: " + triangle.area()); // Output: Triangle Area: 10.5
    }
}
```

3) Collections Framework:

1. What are the differences between ArrayList, LinkedList, and Vector in terms of performance and use cases? When might each be appropriate?
2. How can you implement your own custom collection class in Java, and what considerations should you take into account regarding thread safety and performance?

1. Differences Between ArrayList, LinkedList, and Vector

- **ArrayList**:
  - **Structure**: Resizable array implementation of the List interface.
  - **Performance**:
    - Provides fast random access (O(1) time complexity) due to underlying array structure.
    - Insertion and deletion at the end are O(1) on average, but can be O(n) if resizing occurs. Insertion and deletion in the middle take O(n) time due to shifting elements.
  - **Use Cases**: Best suited for scenarios where frequent access or traversal is needed, and where the frequency of adding/removing elements is relatively low.
- **LinkedList**:
  - **Structure**: Doubly-linked list implementation of the List interface.
  - **Performance**:
    - Provides fast insertion and deletion (O(1) time complexity) at both ends of the list.
    - Random access is slower (O(n)) because it may require traversing the list from the start or the end.
  - **Use Cases**: Suitable for applications that require frequent additions and removals of elements from the start or middle of the list.
- **Vector**:
  - **Structure**: Growable array of objects that is synchronized.
  - **Performance**:
    - Similar to ArrayList in terms of random access, but has a performance overhead due to synchronization (making operations thread-safe).
    - Insertion and deletion performance is also comparable to ArrayList.
  - **Use Cases**: Historically used for thread-safe operations, but typically obsolete now in favor of ArrayList and other collections with explicit synchronization.

2. Implementing a Custom Collection Class

- When implementing a custom collection class, consider the following:
  - **Data Structure**: Choose the underlying data structure carefully (e.g., array, linked list).
  - **Generics**: Use generics to ensure type safety and allow flexibility with the data types stored in the collection.

- **Thread Safety**: Decide if your collection needs to be thread-safe (e.g., using synchronization or concurrent collections).
- **Performance**: Analyze the expected use cases to optimize operations like insertion, retrieval, and deletion.

```
// Demonstrating ArrayList, LinkedList, and Vector
public class CollectionExample {
    public static void main(String[] args) {
        // Use ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("Apple");
        arrayList.add("Banana");
        arrayList.add("Cherry");
        System.out.println("ArrayList: " + arrayList);

        // Use LinkedList
        List<String> linkedList = new LinkedList<>();
        linkedList.add("Dog");
        linkedList.add("Cat");
        linkedList.add("Fish");
        System.out.println("LinkedList: " + linkedList);

        // Use Vector
        List<String> vector = new Vector<>();
        vector.add("Red");
        vector.add("Green");
        vector.add("Blue");
        System.out.println("Vector: " + vector);

        // Custom collection example
        MyCustomCollection<String> myCollection = new MyCustomCollection<>();
        myCollection.add("One");
        myCollection.add("Two");

        System.out.println("Custom Collection: " + myCollection);
        /*ArrayList: [Apple, Banana, Cherry]
        LinkedList: [Dog, Cat, Fish]
        Vector: [Red, Green, Blue]
        Custom Collection: [One, Two]*/
    }
}
// Custom Collection Class
class MyCustomCollection<T> {
    private List<T> elements;

    public MyCustomCollection() {
        elements = new ArrayList<>(); // Using ArrayList as the underlying structure
    }
    public void add(T element) {
        elements.add(element);
    }
    public T get(int index) {
        return elements.get(index);
    }
    public int size() {
        return elements.size();
    }
    @Override
    public String toString() {
        return elements.toString();
    }
}
```

4) Multithreading:

1. What are the different ways to create threads in Java? Compare and contrast implementing Runnable, extending Thread, and using the Executor framework.
2. How do synchronization and locking mechanisms (such as synchronized, ReentrantLock, etc.) influence the design of multithreaded applications?

1.Different Ways to Create Threads:

- **Implementing Runnable**:

This involves creating a class that implements the Runnable interface and overriding the run() method, which contains the code to be executed in the new thread. This approach promotes better separation of concerns and allows for the implementation of multiple interfaces.

- **Extending Thread**:

In this method, you create a class that extends the Thread class and override the run() method. It is a straightforward approach but less flexible compared to implementing Runnable.

- **Using the Executor Framework**:

The Executor framework provides a high-level API for managing threads. It offers a pool of threads, which can be reused, improving performance by reducing thread creation overhead. It allows for more sophisticated thread management and task scheduling.

2.Synchronization and Locking Mechanisms:

- These mechanisms ensure that only one thread can access a resource at a time, preventing data inconsistency.
- synchronized keyword is the simplest way to manage concurrency but can lead to thread contention.
- ReentrantLock allows for more advanced features like timed locks and interruptible locks, providing finer control over thread synchronization.

```java
/**
 * Using class Thread to override run();
 * */
public class MyThread extends Thread{

    private String threadName;

    public MyThread(String threadName) {
        this.threadName = threadName;
    }

    public String getThreadName() {
        return threadName;
    }

    public void setThreadName(String threadName) {
        this.threadName = threadName;
    }

    @Override
    public void run() {
        try {
            System.out.println("MyThread is started!");
```

```
            Thread.sleep(2000);
            System.out.println("MyThread is finished!");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread("myThread");
        myThread.start();
        //change name of thread:
        myThread.setThreadName("updated: myThread");
        System.out.println(myThread.getThreadName());
    }
}

/**
 * Using interface Runnable to implement run();
 * */
public class MyRunnable implements Runnable{

    @Override
    public void run() {
        try {
            System.out.println("MyRunnable class started!");
            Thread.sleep(1000);
            System.out.println("MyRunnable class finished!");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        Thread threadRunnable = new Thread(new MyRunnable());
        threadRunnable.setName("threadRunnable");
        threadRunnable.start();
        System.out.println(threadRunnable.getName());
    }
}
/**
 * Using interface ExecutorService;
 * */
public class MyExecutor {
    public static void main(String[] args) {
        //create pool of threads:
        ExecutorService executor = Executors.newFixedThreadPool(2);
        //submitting task and execute:
        for (int i = 0; i < 10; i++) {
            final int taskID = i;
            executor.submit(()-> System.out.println("[Thread executing using Executor
Framework] task: " + taskID));
        }
        //shutdown executor (will close  and no more tasks available to execute)
        //main thread program is down
        executor.shutdown();
    }
}
```

```
public class Counter {

    private int count = 0;

    public synchronized void increment(){
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class CounterProgram {
    public static void main(String[] args) {
        //create instance of Counter:
        Counter counter = new Counter();
        //creating 3 threads and incrementing value in each:
        //used helper method to get array of threads with counter instance as parameter
        //for each thread we start()
        Thread[] threads = getThreads(counter);
        for (Thread thread : threads) {
            thread.start();
        }
        //joining the started threads (ensure to finish all):
        try {
            for (Thread thread : threads) {
                thread.join();
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        //printing result (expected 3000: 1000 for each)
        System.out.println("Final result: " + counter.getCount());
    }
    //creating threads and incrementing counter:
    //increment method is synchronized (ensure consistency)
    private static Thread[] getThreads(Counter counter) {
        Thread thread1 = new Thread(()->{
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        Thread thread2 = new Thread(()->{
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        Thread thread3 = new Thread(()->{
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        //return threads array
        Thread[] threads = {thread1, thread2, thread3};
        return threads;
    }


}
```

5) Best Practices in Multithreading:

1. What best practices should be followed when working with collections in a multithreaded environment to avoid common pitfalls such as concurrent modification exceptions?
2. How can you effectively manage resources and minimize contention in a multithreaded Java application?

1. **Best Practices for Collections in a Multithreaded Environment**:

Use Concurrent Collections: Java provides concurrent collection implementations like ConcurrentHashMap, CopyOnWriteArrayList, and BlockingQueue that are designed to handle concurrent access safely without extensive locking.

Avoid Modifying Collections While Iterating: When iterating over collections, avoid structural modifications (adding/removing elements) which can lead to ConcurrentModificationException. Instead, use an iterator's remove method or copy the collection to a new one for modifications.

Immutable Collections: If possible, use immutable collections (e.g., those from Collections.unmodifiableList()) to prevent changes from being made by other threads, ensuring safe access.

Fine-Grained Locking: If using traditional collections, consider locking specific resources instead of the entire collection, allowing for higher concurrency.

2. **Managing Resources and Minimizing Contention**:

Limit Shared State: Reduce shared mutable state among threads. Where possible, design classes to be thread-safe and encapsulate shared resources

Use Thread Pools: Manage the lifecycle of threads efficiently using thread pools, as it reduces overhead and improves application performance.

Use Timeouts: When acquiring locks, use a timeout mechanism to avoid deadlocks and keep the application responsive.

Profile and Monitor: Regularly profile and monitor the application to identify bottlenecks and areas of contention, thus allowing for informed optimization.

```java
/**
 * Using Concurrent Collections in example of ConcurrentHashMap;
 * */
public class MainConcurrentHashMap {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

        //add elements:
        map.put("One", 1);
        map.put("Two", 2);
        map.put("Three", 3);
        map.put("Four", 4);
        map.put("Five", 5);

        //check status before:
        System.out.println("before modification map:");
        map.forEach((s, integer) -> System.out.print("[" + s + " : " + integer + "] "));
        System.out.println("\nMap SIZE (before): " + map.size());

        //check status after:
        System.out.println("after concurrently modification:");
        // Concurrently modifying the map
```

```
        map.forEach((key, value) -> {

            if (key.equals("Five-NotExistKey")) {
                map.put("Six", 6);
            }
            if (key.equals("Five")) { //if such key exist - if statement will execute
                map.put("Six", 6); // No ConcurrentModificationException
            }
            if (key.equals("Six")) {
                map.put("Seven", 7);
            }
            if(key.equals("One")){
                map.put("Eight", 8);
            }
            System.out.print("[" + key + ": " + value + "] "); //not all keys is viewed

        });
        System.out.println("\nMap SIZE (after): " + map.size()); //but size is correct

        System.out.println("***final result:***");
        map.forEach((s, integer) -> System.out.print("[" + s + " : " + integer + "]"));
//we could see all keys
        System.out.println("\nMap SIZE (after): " + map.size());

        /*      before modification map:
            [Five : 5] [One : 1] [Four : 4] [Two : 2] [Three : 3]
            Map SIZE (before): 5

                after concurrently modification:
            [Five: 5] [Six: 6] [One: 1] [Four: 4] [Seven: 7] [Two: 2] [Three: 3]
            Map SIZE (after): 8

                ***final result:***
            [Eight : 8][Five : 5][Six : 6][One : 1][Four : 4][Seven : 7][Two : 2][Three :
3]
            Map SIZE (after): 8         */

    }
}
```

```
public class AvoidConcurrentModification {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Pineapple");
        fruits.add("Banana");

        for (String fruit : fruits) {
            System.out.print(fruit + " "); //Apple Pineapple Banana
        }
        System.out.println();

        // Create a copy of the list (fruits) for safe modification:
        List<String> copyFruits = new ArrayList<>(fruits);
        for (String copyFruit : copyFruits) {
            if(copyFruit.equals("Banana")){
                fruits.remove(copyFruit); //no ConcurrentModificationException
            }
        }
```

```
        //check the original fruits list:
        System.out.println(fruits); //[Apple, Pineapple]
        /**
         * Description:
         *
         * Copying the List:
         * You create a new list called copyFruits using the constructor new
ArrayList<>(fruits),
         * which creates a shallow copy of the original fruits list. This means that
copyFruits contains references
         * to the same elements as fruits, but it is a different list.
         *
         * Iteration and Modification:
         * When you iterate over copyFruits, you check if any of the elements equal
"Banana". If a match is found,
         * you remove this element from fruits. Since you're modifying the fruits list
during the iteration of copyFruits,
         * no ConcurrentModificationException is thrown because you are not directly
modifying the list that you are iterating over.
         * You are modifying a different list (fruits) based on the conditions checked in
copyFruits.
         *
         * Effects of Modification:
         * Because copyFruits is a shallow copy that references the same String objects
(the actual fruit names are immutable),
         * removing "Banana" from fruits will reflect in the original list because the
reference is the same.
         * However, the structure of copyFruits remains intact, allowing the program to
continue iterating safely.
         * */

    }
}
```

```
public class MainUsingThreadPool {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 7; i++) {
            final int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " is running");
                // Simulate work
                try {
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        executor.shutdown();
    }
}
```

6) Real-World Applications:

1. Give examples of real-world scenarios where multithreading is essential. How does Java handle multithreading in environments such as web servers or data processing applications?
2. How can understanding type safety and polymorphism lead to better APIs and frameworks that are robust and easier to understand?

1. **Examples of Real-World Scenarios Where Multithreading is Essential**:

   - **Web Servers**: Multithreading allows web servers to handle multiple requests simultaneously. Each incoming request can be processed in a separate thread without blocking the server from accepting new requests, leading to better resource utilization and faster response times.
   - **Data Processing Applications**: In applications that need to process large datasets—like ETL (Extract, Transform, Load) processes—multithreading can enable parallel processing of data, significantly reducing the time required for operations such as data transformation or aggregation.
   - **User Interface Applications**: In GUI applications, multithreading is crucial to keep the user interface responsive while performing long-running background tasks (like file downloads, database queries, etc.)—by using worker threads to handle these tasks.

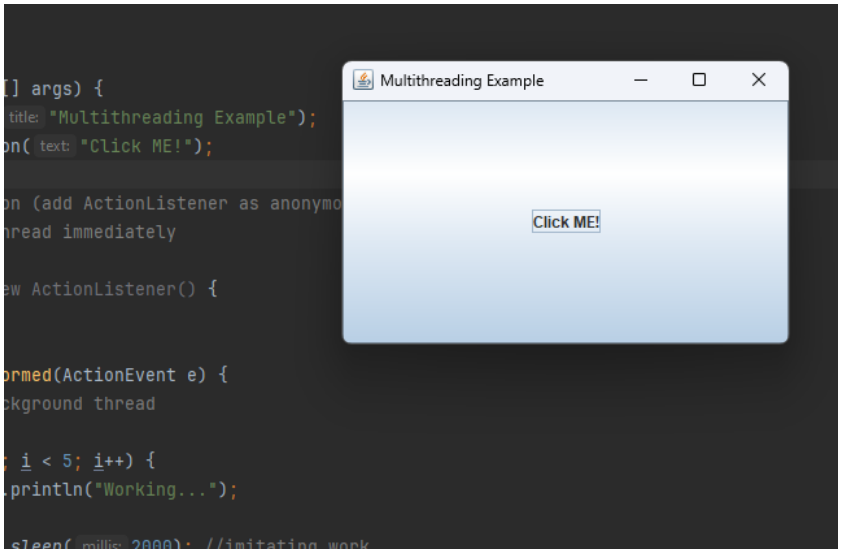
2. **Java's Handling of Multithreading in Environments**:

   - Java provides robust support for multithreading and concurrent programming through its java.lang package and the java.util.concurrent package, which includes higher-level abstractions such as executors, concurrent collections, and synchronization utilities. The Java Virtual Machine (JVM) manages threads efficiently, allowing developers to focus on application logic rather than thread management.

```
public class SimpleWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8080);
        System.out.println("Server started and listening on port: 8080");

        while (true) {
            Socket clientSocket = serverSocket.accept();
            new Thread(new ClientHandler(clientSocket)).start();
        }
        //print on browser: http://localhost:8080/
        //and get response: Hello, World!

    }
}
public class ClientHandler implements Runnable{

    private Socket clientSocket;

    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }

    @Override
    public void run() {
        try (BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)) {
            String requestLine = in.readLine();
```

```
            // Process the request (omitting details for brevity)
            out.println("HTTP/1.1 200 OK");
            out.println("Content-Type: text/plain");
            out.println();
            out.println("Hello, From Client Server!");
        } catch (IOException e)
        {
            e.printStackTrace();
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class UISample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Multithreading Example");
        JButton button = new JButton("Click ME!");

        //creating action for button (add ActionListener as anonymous class and Thread
inside also anonymous)
        //after creating started thread immediately
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                //start task in background thread
                new Thread(()->{
                    for (int i = 0; i < 5; i++) {
                        System.out.println("Working...");
                        try{
                            Thread.sleep(2000); //imitating work
                        }catch (InterruptedException ex){
                            Thread.currentThread().interrupt();
                        }
                    }
                }).start();
            }
        });

        //setting for frame:
        frame.getContentPane().add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```




```java
/**
 * This AdvancedDataProcessor class demonstrates multithreading with more complex logic
while maintaining
 * thread safety and providing clear feedback about the computations performed.
 * */
public class AdvancedDataProcessor {
    private static final Logger log =
Logger.getLogger(String.valueOf(AdvancedDataProcessor.class));
    private static final AtomicInteger totalSumOfSquares = new AtomicInteger(0);
    private static final AtomicInteger totalSumOfQubes = new AtomicInteger(0);

    public static void main(String[] args) {
        //executor created:
        ExecutorService executorService = Executors.newFixedThreadPool(4);

        //store for futures results of calculating data:
        List<Future<String>> results = new ArrayList<>();

        //initiating and submit tasks for calculating:
        for (int i = 0; i < 10; i++) {
            final int taskID = i;

            results.add(executorService.submit(()->{
                try {
                    //perform calculations:
                    int square = calculateSquare(taskID);
                    int qube = calculateQube(taskID);

                    //update totals in thread-safe manner:
                    totalSumOfSquares.addAndGet(square);
                    totalSumOfQubes.addAndGet(qube);

                    return "Task: " + taskID + ", processed:  Square [" + square + "]; "
+ " Qube [" + qube + "]";

                } catch (Exception e) {
                    log.severe("TaskID: " + taskID + " encountered an error: " +
e.getMessage());
                    return "Task ID: " + taskID + " encountered an error: " +
```

```
e.getMessage();
                }
            }));

        }

        //printing results
        for (Future<String> result : results) {
            try {
                System.out.println(result.get());
            }catch (Exception e){
                log.severe("Error retrieving result: " + e.getMessage());
            }
        }

        //print total sums:
        System.out.println("Total sum of Squares: " + totalSumOfSquares.get());
        System.out.println("Total sum of Qubes: " + totalSumOfQubes.get());

        //end of work and shutdown executor:
        executorService.shutdown();
    }

    private static int calculateSquare(int number){
        try {
            Thread.sleep(200);
        }catch (InterruptedException e){
            Thread.currentThread().interrupt();
        }
        return number * number;
    }

    private static int calculateQube(int number){
        try {
            Thread.sleep(200);
        }catch (InterruptedException e){
            Thread.currentThread().interrupt();
        }
        return number * number * number;
    }
}

/*Task: 0, processed:  Square [0];  Qube [0]
Task: 1, processed:  Square [1];  Qube [1]
Task: 2, processed:  Square [4];  Qube [8]
Task: 3, processed:  Square [9];  Qube [27]
Task: 4, processed:  Square [16];  Qube [64]
Task: 5, processed:  Square [25];  Qube [125]
Task: 6, processed:  Square [36];  Qube [216]
Task: 7, processed:  Square [49];  Qube [343]
Task: 8, processed:  Square [64];  Qube [512]
Task: 9, processed:  Square [81];  Qube [729]
Total sum of Squares: 285
Total sum of Qubes: 2025*/
```

7) Functional Programming:

1. Discuss how the introduction of streams and lambda expressions in Java 8 has changed the way we work with collections and how it relates to polymorphism.
2. Explore how design patterns, like the Factory or Visitor patterns, can leverage polymorphism and type safety in their implementations.

1. **Streams and Lambda Expressions in Java 8**:

   - **Lambda Expressions**: Introduced in Java 8, lambda expressions allow for a more concise way to express instances of single-method interfaces (functional interfaces). They simplify the code by reducing boilerplate code associated with anonymous classes.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name)); // Using lambda expression
```

   - **Streams**: A major addition in Java 8, streams allow for functional-style operations on collections. A stream abstracts the iteration and provides a high-level way to process sequences of elements (such as filtering, mapping, and reducing).

```
List<String> filteredNames = names.stream()
        .filter(name -> name.startsWith("A")) // Filtering names starting with A
        .collect(Collectors.toList()); // Collecting results back to a list
```

2. **Relationship to Polymorphism**:

   - Polymorphism in the context of functional programming is heavily utilized through the use of functional interfaces (like Predicate, Function, Consumer, etc.). Lambda expressions are a way to implement these interfaces, allowing for more dynamic behavior.

   - Because functional interfaces can be treated as first-class citizens, we can pass them as arguments, store them in variables, and return them from methods, enhancing the ability to create flexible APIs that can interact with various types in a type-safe manner.

Visitor Pattern

```
public class VisitorPatternExample {
    public static void main(String[] args) {
        // Create shapes
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Circle());
        shapes.add(new Rectangle());

        // Create visitor
        AreaCalculator areaCalculator = new AreaCalculator();

        // Calculate area using visitor
        for (Shape shape : shapes) {
            shape.accept(areaCalculator); // Each shape accepts the visitor
        }
        // Output the total area
        System.out.println("Total Area: " + areaCalculator.getTotalArea());
    }
    /**
     * Explanation of the Visitor Pattern:
     *
```

```
     * Polymorphism:
     * The Visitor interface declares different visit methods for every shape type,
allowing the AreaCalculator
     * to operate on various shape types without needing to know their details.
Each shape implements the accept method that
     * takes a Visitor and calls the appropriate visit method based on its type.
     *
     * Extensibility:
     * Adding a new shape class (e.g., Triangle) would require only implementing
the Shape interface and adding a new
     * method in the Visitor class, thus benefiting from the Open/Closed Principle.
     *
     * Decoupling:
     * The operations on the objects are encapsulated in the visitor classes,
separating the algorithm
     * from the object structure, which enhances maintainability.
     * */
}
  class AreaCalculator implements Visitor {
          private double totalArea = 0;

          @Override
          public void visit(Circle circle) {
         // Sample calculation for Circle with radius 1
         double radius = 1; // Hardcoded for simplicity
         totalArea += Math.PI * radius * radius; // Area = πr²
          }

    @Override
    public void visit(Rectangle rectangle) {
        // Sample calculation for Rectangle with width 2 and height 3
        double width = 2; // Hardcoded for simplicity
        double height = 3; // Hardcoded for simplicity
        totalArea += width * height; // Area = width * height
    }

    public double getTotalArea() {
        return totalArea;
    }
}
interface Visitor {
    void visit(Circle circle);    // Visit method for Circle
    void visit(Rectangle rectangle); // Visit method for Rectangle
}
interface Shape {
    void accept(Visitor visitor); // Accepts a visitor
}
class Rectangle implements Shape {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this); // Accept the visitor
    }
}
class Circle implements Shape {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this); // Accept the visitor
    }
}
```

Factory Pattern

```
/**
 * The Factory Pattern leverages polymorphism by returning different types of objects
that share a common interface.
 * This allows for the creation of families of related objects without specifying their
concrete classes, enhancing type safety and flexibility.
 * */
class ShapeFactory {
    public Shape createShape(String shapeType) {
        switch (shapeType.toLowerCase()) {
            case "circle":
                return new Circle();
            case "rectangle":
                return new Rectangle();
            default:
                throw new IllegalArgumentException("Unknown shape type");
        }
    }
}
interface Shape {
    void draw();
}
class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
```

8) Comparative Analysis:

- Discuss the strengths and weaknesses of the Collections Framework in Java compared to other languages (like C#, Python) in terms of type safety and data handling.
- Explore the impact of using different collection types on application performance for various scenarios, including heavy read/write operations.

1.Strengths and Weaknesses of the Collections Framework:

Java:

- **Strengths**:

Type Safety: Java's collections are generics-based, providing compile-time type checking and minimizing runtime errors related to type casting.

Concurrent Collections: Java provides a rich set of concurrent collections
(e.g., ConcurrentHashMap, CopyOnWriteArrayList) that facilitate safe access from multiple threads.

Rich API: The Collections Framework offers a wide range of data structures (lists, sets, maps, queues) along with various utility methods and a solid iterator mechanism.

- **Weaknesses**:

Verbosity: Java's syntax can be more verbose compared to Python and C#, leading to more boilerplate code.

Performance Overhead: The use of generics and the need for boxing/unboxing with primitives can introduce some performance overhead.

C#:

- **Strengths**:

LINQ Support: C# provides Language Integrated Query (LINQ), which allows for complex querying directly on collections, enabling concise and readable code for data manipulation.

Flexible Collections: The .NET Framework offers a rich set of collection types, including lists, dictionaries, and concurrent collections.

- **Weaknesses**:

Type Inference: While C# has generics, it can sometimes result in less strict type checks compared to Java due to the use of type inference with var.

Python:

- **Strengths**:

Dynamic Typing: Python's collections (like lists and dictionaries) enjoy dynamic typing, allowing more flexibility and ease of use, making them exceptionally easy to work with for rapid development.

Rich Built-in Data Types: Python natively supports a wide variety of data structures, such as lists, tuples, sets, and dictionaries, with simple and clean syntax.

- **Weaknesses**:

Lack of Type Safety: The dynamic nature leads to potential runtime errors that would be caught at compile time in statically typed languages like Java or C#.

Performance: Python's interpreted nature may lead to slower performance for certain operations compared to compiled languages.

2.Impact of Different Collection Types on Application Performance:

Heavy Read Operations:

Java: Use ArrayList or HashMap for efficient accessing and iteration, but be cautious of concurrency issues in multithreaded environments.

C#: Collections like List<T> or Dictionary<TKey, TValue> perform well in read-heavy applications, and using ConcurrentDictionary<TKey, TValue> can provide thread-safe reads with minimal contention.

Python: Native lists and dictionaries are highly optimized. Using a deque from the collections module can improve performance for large lists with frequent append and pop operations.

Heavy Write Operations:

Java: For frequent write operations, using LinkedList allows for O(1) insertions/removals but lacks random access efficiency.

C#: The List<T> class performs reasonably well. However, when performance is crucial during frequent insertions, LinkedList<T> may be more suitable.

Python: Lists allow for quick appends but can be slow for insertions in the middle. Utilizing a deque can provide better performance for situations involving frequent additions or deletions at both ends.

Concurrent Operations:

Java: Use concurrent collections designed for safe access from multiple threads; this provides built-in mechanisms to handle locks and minimize contention.

C#: The ConcurrentQueue<T> and ConcurrentBag<T> are specifically designed for concurrent operations, providing thread safety with lower performance cost.

Python: Python's Global Interpreter Lock (GIL) often limits thread-based concurrency; using multiprocessing modules or thread-safe queues (like queue.Queue) is preferable for concurrent tasks.

Summary: Each language has its strengths and weaknesses when it comes to collection handling, type safety, and performance under different conditions. Java provides a robust framework with strong type safety and concurrent collections but at the cost of verbosity. C# offers powerful querying capabilities through LINQ and rich collection types but can be less strict with type inference. Python, with its dynamic typing and rich built-in data types, excels in ease of use and rapid development but lacks the type safety that Java and C# provide.