12.4-Runnable vs Thread

Introduction to the Runnable Interface

The Runnable interface in Java is a functional interface that contains a single abstract method, **run()**. Because it only has one method, you often use it in scenarios where you want to execute code concurrently in a separate thread without inheriting from the Thread class.

Why Use Runnable?

When we want to create a new thread, we usually call the start() method. Internally, the **start()** method of the Thread class which invokes the run() method, and it does so by expecting a Runnable instance. This allows us to decouple the task logic from the actual thread implementation.

Example Scenario: Multiple Inheritance Issue

Let's consider a scenario where you want to create a class Z as the parent class for class A, but you also want A to perform some concurrent operations using the Thread class. However, Java does **not** support multiple inheritance, so you cannot extend both Thread and Z at the same time.

Incorrect Example:

```
class Z { }
class A extends Z, Thread {  // Not allowed in Java as it does not
support multiple inheritance
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hi, I'm from class A");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e);
            }
        }
    }
}
```

Solution: Using the Runnable Interface

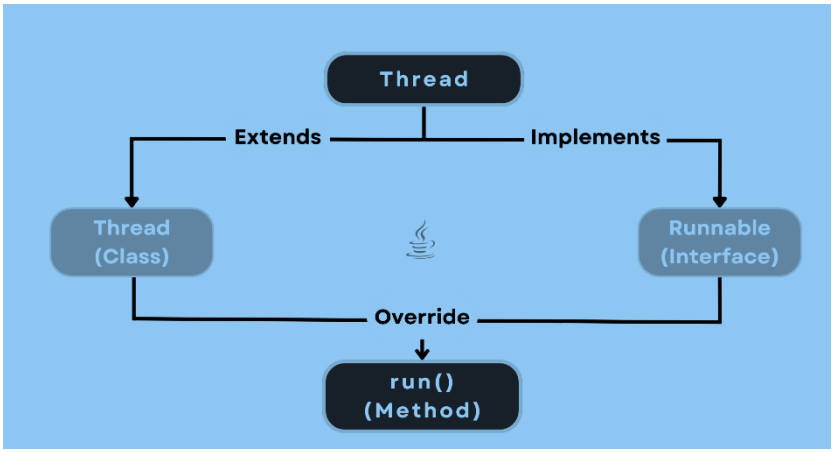
Instead of extending Thread, we implement the Runnable interface. This way, we can still extend the class Z while also implementing Runnable to achieve the desired functionality.

```
class Z { }

class A extends Z implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hi, I'm from class A");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e);
            }
        }
    }
}
```

Runnable vs. Thread: A Better Approach

The above implementation allows class A to extend its parent class Z and simultaneously perform tasks in a new thread by implementing Runnable also It is important to note that calling start() initiates a new thread, which in turn calls the run() method. Directly invoking run() will execute the task sequentially in the main thread, contradicting the purpose of multithreading.
Let's extend the example to demonstrate the use of two classes implementing Runnable:



Example: Two Runnable Implementations

```
class A implements Runnable {
        public void run() {
    for (int i = 0; i < 5; i++) {
System.out.println("Hi, I'm from class A");
                    try {
                Thread.sleep(1000);
    } catch (InterruptedException e) {
System.out.println("Thread interrupted: " + e);
                    }
                }
            }
        }

class B implements Runnable {
        public void run() {
    for (int i = 0; i < 5; i++) {
System.out.println("Hello, I'm from class B");
                    try {
                Thread.sleep(1000);
    } catch (InterruptedException e) {
System.out.println("Thread interrupted: " + e);
                    }
                }
            }
        }

        public class Demo {
    public static void main(String[] args) {
            Runnable obj1 = new A();
            Runnable obj2 = new B();

Thread t1 = new Thread(obj1);  // Creating thread t1
Thread t2 = new Thread(obj2);  // Creating thread t2

        t1.start();  // Start thread t1
        t2.start();  // Start thread t2
                }
            }
```

Output:

```
Hi, I'm from class A
Hello, I'm from class B
Hi, I'm from class A
Hello, I'm from class B
...
```

Explanation:

1. The class A and B implement the Runnable interface.
2. We create two separate threads (t1 and t2) using instances of Runnable implementations (obj1 and obj2).
3. This approach allows us to have more flexible designs by decoupling the thread execution from class inheritance.

Optimizing Code with Anonymous Classes and Lambda Expressions

Sometimes, creating separate classes just to implement the Runnable interface can make the code lengthy and less readable. To solve this, we can use **Anonymous Classes** and **Lambda Expressions** to optimize our code.

Using Anonymous Classes

```
public class Demo {
    public static void main(String[] args) {
        Runnable obj1 = new Runnable() {
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("Hi, I'm from class A");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        System.out.println("Thread interrupted: " +
e);
                    }
                }
            }
        };

        Runnable obj2 = new Runnable() {
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("Hello, I'm from class B");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        System.out.println("Thread interrupted: " +
```

```
e);
                    }
                }
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

        t1.start();
        t2.start();
    }
}
```

Using Lambda Expressions

Since Runnable is a functional interface, we can simplify the code using **lambda expressions**:

```
public class Demo {
    public static void main(String[] args) {
        Runnable obj1 = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Hi, I'm from class A");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Thread interrupted: " + e);
                }
            }
        };

        Runnable obj2 = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Hello, I'm from class B");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Thread interrupted: " + e);
                }
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);
```

```
        t1.start();
        t2.start();
    }
}
```

Explanation of Lambda Optimization:

- **Lambda expressions** reduce code verbosity and make the implementation more concise.
- This approach is highly recommended when implementing single-method interfaces like Runnable.

Differences Between Thread and Runnable

1. Extending vs. Implementing:

- **Thread**: If a class extends Thread, it cannot extend any other class due to Java's single inheritance constraint.
- **Runnable**: A class can implement multiple interfaces, allowing more flexibility.

2. Reusability:

- **Thread**: The task and the thread are tightly coupled, which can limit reusability.
- **Runnable**: Separates the task from the thread, promoting better object-oriented design and modularity.

3. Resource Sharing:

- **Runnable**: Promotes resource sharing as a single Runnable instance can be shared among multiple threads.
- **Thread**: Each thread has its own instance, which can lead to increased memory consumption.

Conclusion

In summary, both the Thread class and the Runnable interface play crucial roles in Java's multithreading model. Developers should carefully evaluate their

application requirements to decide between the two. While Thread offers straightforward usage, Runnable is typically preferred due to its flexibility, reusability, and better support for resource sharing.