Future

Java Future: A Detailed Lesson

In Java, `Future` is part of the `java.util.concurrent` package and is used to represent the result of an asynchronous computation. It allows you to perform long-running tasks in the background and retrieve the result at a later time. The `Future` interface provides methods to check if the task is complete, wait for the task to finish, and retrieve the result when it's available.

---

1. What is `Future`?

`Future` represents the result of an asynchronous operation that may or may not have completed yet. It provides the following key capabilities:

- **Asynchronous Execution**: It can run tasks in the background without blocking the main thread.
- **Task Status Check**: You can check whether the task is done or still running.
- **Retrieve Result**: You can retrieve the result when the task is complete.
- **Cancel Task**: You can attempt to cancel the task if needed.

---

2. Key Methods in the `Future` Interface

Method	Description
`boolean cancel(boolean mayInterrupt)`	Attempts to cancel the task.
`boolean isCancelled()`	Returns `true` if the task was cancelled before completion.
`boolean isDone()`	Returns `true` if the task has completed, either normally or through cancellation.
`T get()`	Retrieves the result, waiting if necessary for the task to complete.
`T get(long timeout, TimeUnit unit)`	Retrieves the result, waiting up to the specified timeout.

---

3. Using `Future` with `ExecutorService`

Typically, a `Future` object is used with an `ExecutorService` to submit tasks for asynchronous execution.

Example: Submitting a Task with `ExecutorService`

```
1  import java.util.concurrent.Callable;
2  import java.util.concurrent.ExecutorService;
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.Future;
5
6  public class FutureExample {
7      public static void main(String[] args) {
```

```
        // Create a thread pool with one thread
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Define a task (Callable) that returns a result
        Callable<Integer> task = () -> {
            System.out.println("Task started...");
            Thread.sleep(2000); // Simulate long-running operation
            return 42;
        };

        // Submit the task to the ExecutorService and get a Future object
        Future<Integer> future = executor.submit(task);

        // Do something else while the task is running...
        System.out.println("Doing other work...");

        try {
            // Retrieve the result (blocks until the task is complete)
            Integer result = future.get();
            System.out.println("Task completed with result: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Shutdown the executor
        executor.shutdown();
    }
}
```

Explanation:

- **Callable Task**: We define a `Callable` task that returns a result (in this case, the number `42`). Unlike `Runnable`, `Callable` can return a result or throw an exception.
- **ExecutorService**: We submit the task using `ExecutorService`, which runs it asynchronously. The `Future` returned by `submit()` represents the eventual result of the task.
- **Future Methods**:
  - `future.get()`: Blocks the main thread until the task is complete and returns the result.
  - `future.isDone()`: Can be used to check if the task is complete before calling `get()`.

---

4. Using `Future.get()` Without Blocking

If you want to avoid blocking the main thread while waiting for the result, you can periodically check whether the task is done using `isDone()` or set a timeout for `get()`.

Example: Polling with `isDone()`

```
import java.util.concurrent.*;

public class FutureNonBlockingExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<Integer> task = () -> {
            Thread.sleep(3000); // Simulate a long-running task
            return 100;
```

```
        };

        Future<Integer> future = executor.submit(task);

        // Polling to avoid blocking the main thread
        while (!future.isDone()) {
            System.out.println("Task is still running...");
            Thread.sleep(500); // Poll every 500ms
        }

        System.out.println("Task completed with result: " + future.get());

        executor.shutdown();
    }
}
```

Explanation:

- **Polling**: Instead of immediately calling `get()`, the code periodically checks `isDone()` to see if the task is complete. Once complete, it calls `get()` to retrieve the result.

---

5. Timeout with `Future.get(long timeout, TimeUnit unit)`

Sometimes you may want to set a time limit on how long you're willing to wait for a result. You can use `get(long timeout, TimeUnit unit)` to specify a timeout.

Example: Using Timeout in `get()`

```
import java.util.concurrent.*;

public class FutureTimeoutExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<Integer> task = () -> {
            Thread.sleep(3000); // Simulate a long-running task
            return 50;
        };

        Future<Integer> future = executor.submit(task);

        try {
            // Wait for at most 1 second to get the result
            Integer result = future.get(1, TimeUnit.SECONDS);
            System.out.println("Task completed with result: " + result);
        } catch (TimeoutException e) {
            System.out.println("Task timed out!");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        executor.shutdown();
    }
}
```

Explanation:

- **TimeoutException**: If the task takes longer than the specified timeout (1 second in this example), a `TimeoutException` is thrown, allowing you to handle the situation gracefully.

---

6. Cancelling a Future Task

You can cancel a running task if it's no longer needed, using the `cancel()` method. This can interrupt a thread if needed.

Example: Canceling a Task

```java
import java.util.concurrent.*;

public class FutureCancelExample {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<Integer> task = () -> {
            Thread.sleep(5000); // Simulate long task
            return 99;
        };

        Future<Integer> future = executor.submit(task);

        Thread.sleep(2000);  // Wait for 2 seconds before canceling
        System.out.println("Canceling task...");
        boolean cancelled = future.cancel(true);  // Attempts to cancel the task

        if (cancelled) {
            System.out.println("Task was cancelled.");
        } else {
            System.out.println("Task was not cancelled.");
        }

        executor.shutdown();
    }
}
```

Explanation:

- **cancel()**: Attempts to cancel the task. If the task is already completed, it cannot be canceled.
- **true in cancel(true)**: If `true`, the thread running the task is interrupted. If `false`, it will not be interrupted but the task won't be executed if it hasn't started yet.
- **isCancelled()**: Can be used to check if the task was successfully canceled.

---

7. Limitations of `Future`

While `Future` is useful, it has some limitations:

- **No Explicit Completion Callbacks**: There is no built-in mechanism to perform an action when the task completes (e.g., notify when a task is done).
- **Blocking Nature**: The `get()` method blocks until the result is available unless you poll or set timeouts.

To overcome these limitations, Java introduced `CompletableFuture` in Java 8, which builds on `Future` and provides a more flexible, non-blocking approach.

---

8. Conclusion

`Future` in Java is a useful construct for managing asynchronous tasks. It allows you to:

- Submit tasks to be executed in the background.
- Retrieve results when the task is done.
- Set time limits and handle cancellation of tasks.

However, in modern applications, `CompletableFuture` offers a more robust and feature-rich alternative to `Future` for handling asynchronous tasks.

Would you like to explore `CompletableFuture` or go deeper into any of these topics?