Technische Hochschule Rosenheim

Modul- Fortgeschrittene Programmierkonzepte

Bachelor Informatik

11- Futures

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Threads

```
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello from my custom thread!");
    }
});

t.start();
System.out.println("Hello from the application main thread!")

System.out.println("Waiting for thread to complete...");
t.join();

System.out.println("All done.");
```

Futures, Callables, Executor

```
interface Callable<V> {
    V call();
}
```

```
interface Future<T> {
    T get();
    // ...
}
```

```
interface ExecutorService {
    void execute(Runnable command);
    <T> Future<T> submit(Callable<T> task);
    // ...
}
```

`ExecutorServices` provided by Java:

```
Executors.newSingleThreadExecutor();
Executors.newCachedThreadPool();   // reuses threads
Executors.newFixedThreadPool(5);   // use 5 threads
```

Chaining with CompletableFuture

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {
    static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
        // ...
    }
    <U> CompletionStage<U> thenApplyAsync(Function<? super T, ? extends U> f
        // ...
    }
    <U> CompletionStage<U> thenAcceptAsync(Consumer<? super T> action) {
        // ...
    }
    CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn)
        // ...
    }
    // and much more...
}
```

Chaining with CompletableFuture

```
CompletableFuture<?> cf = CompletableFuture.supplyAsync(() -> "riko493:12345
    .thenApplyAsync(creds -> {
        System.out.println("Authenticating with " + creds);
        return "secrettoken";
    })
    .thenApplyAsync(token -> {
        System.out.println("Retrieving status with token=" + token);
        return "in the mood for holidays";
    })
    .thenAccept(status -> System.out.println(status))
    .exceptionally(ex -> { System.out.println("Oops, something went wrong: "

System.out.println("All done!");
```

Summary (1)

Technische Hochschule Rosenheim

Use threads to run code asynchronously and in parallel.

- *asynchronously* means the caller/delegator immediately continues execution after queueing the task
- *parallel* means that more than one method is executed at the same time; asynchronous methods are typcially executed in parallel.
- use `join` to wait on threads to complete

Be extra careful when threads *share resources*.

- use thread-safe containers
- protect critical sections with `synchronized`
- minimize blocking time and avoid deadlocks with inter-thread communication (`wait`, `yield`, `notify`, `notifyAll`)

Summary (2)

Technische Hochschule Rosenheim

Use `Future` to retrieve results and handle exceptions within the threads.

- `get()` will block until the thread has completed (and raise possible exception)
- `cancel()` terminates the execution of the task
- `isDone()` returns `true` if the task completed

Use `CompletableFuture` for elegant asynchronous programming.

- use `supplyAsync` to create a chainable `CompletableFuture`
- use `thenApplyAsync` to transform intermediate results
- use `thenAcceptAsyc` to consume final results (end of chain!)
- use `thenCombineAsync` to join multiple `CompletableFuture`

Use `ExecutorServices` when batch-processing large quantities of data, e.g. importing multiple files, scaling images, downloading resources, etc.

holidays

Enjoy the holiday break!