Concurrency

Using Concurrency API

Concurrency API

- can be used by importing `java.util.concurrent` package
- this package includes `ExecutorService` interface
  - this interface defines services which create and manage threads
  - includes features like thread pooling, thread scheduling, etc.

```
// single-thread executor
ExecutorService service = Executors.newSingleThreadExecutor();

service.execute(thread1);
service.execute(thread2);
service.execute(thread3);

service.shoutdown();
```

threads are executed one by one

if omitted, the program will never end

Future<V> instance

- there are two ways you can execute `Runnable` task
  1. using `execute(Runnable task)` method
  2. using `submit(Runnable task)` method
- the difference is that `submit()` **returns a value**
  - this value is instance of a special interface called `Future<V>`
  - this instance can be used to determine the result of the execution

Future<V> interface methods

`boolean isDone()`

- returns `true` if task was completed, threw exception or was cancelled

`boolean isCanceled()`

- returns `true` if task was canceled before completed normally

`boolean cancel(boolean mayInterruptIfRunning)`

- attempts to cancel the execution of the task, returns `true` if it was canceled

`V get()`

- retrieves the result of the task

`V get (long timeout, TimeUnit unit)`

- retrieves the result of the task, waiting specified amount of time
- if the result is not ready by that time, checked `TimeoutException` will be thrown

TimeUnit values

```
TimeUnit.NANOSECONDS

TimeUnit.MICROSECONDS

TimeUnit.MILLISECONDS

TimeUnit.SECONDS

TimeUnit.MINUTES

TimeUnit.HOURS

TimeUnit.DAYS
```

```
ExecutorService service = Executors.newSingleThreadExecutor();

Future<?> result = service.submit(() -> {
  for (int i = 0; i < 10_000_000; i++) count++; });
```

a process which takes several milliseconds to complete

```
try {
   var value = result.get(1, TimeUnit.MILLISECONDS);
   if (value == null) System.out.println("Task completed.");
} catch (TimeoutException e) {
   System.out.println("Task didn't complete in time.");
} catch (InterruptedException | ExecutionException e) {
   e.printStackTrace();
}


service.shutdown();
```

result of the task after 1 ms

task was not completed in 1 ms

Callable Interface

- similar to `Runnable`, except:
  - method you need to implement is called `call()`
  - `call()` method **returns a value** and can throw a checked exception
- `ExecutorService` includes overloaded version of the `submit()` method
  - you can pass `Callable` object to `submit()` and get `Future<T>` instance
- when passing `Runnable`, `get()` returns null if the task is complete
  - with `Callable`, `get()` returns the matching generic type

```
var service = Executors.newSingleThreadExecutor();
try {
    Future<Integer> result = service.submit(() -> 11 * 12);
    System.out.println(result.get());
} finally {
    service.shutdown();
}


=> 132


// Q. How does Java know if you've passed the implementation for run() or call()?
// A. It knows because run() doesn't return a value, and call() does.
```

implementation of `call()` method

returns `Integer` result

Scheduling Tasks

- to schedule tasks we use `ScheduledExecutorService` interface with:

`schedule(Callable<V> callable, long delay, TimeUnit unit)`

creates and executes Callable task after given delay

`schedule(Runnable task, long delay, TimeUnit unit)`

creates and executes Runnable task after given delay

`scheduleAtFixedRate(Runnable task, long initDelay, long period, TimeUnit unit)`

creates and executes Runnable task after initial delay and creating new task every period value that passes

`scheduleWithFixedDelay(Runnable task, long initDelay, long period, TimeUnit unit)`

creates and executes Runnable task after initial delay and subsequently with given delay between termination of on and execution of the next one

```
ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();

Runnable taskOne = () -> System.out.println("Hello!");
Callable<String> taskTwo = () -> "Hi!";

ScheduledFuture<?> resultOne = service.schedule(taskOne, 20, TimeUnit.SECONDS);
ScheduledFuture<?> resultTWo = service.schedule(taskTwo, 15, TimeUnit.MINUTES);

// taskOne is scheduled 20 seconds in the future

// taskTwo is scheduled 15 minutes in the future
```

Scheduling Thread Pool

- thread pool is a group of pre-instantiated reusable threads
  - available to perform a set of arbitrary tasks

`ExecutorService newCachedThreadPool()`

creates thread pool that creates new threads as needed, but reuses previously constructed threads when they are available

`ExecutorService newFixedThreadPool(int noOfThreads)`

creates thread pool that reuses fixed number of threads operating off shared unbounded queue

`ScheduledExecutorService newScheduledThreadPool(int noOfThreads)`

creates thread pool that can schedule commands to run after given delay or execute periodically