# Task & Task Management

- Task: a unit of work

- Task management: manipulating & keeping track of a set of tasks

# Task Representation

- Java task representations:

  1. Interface `java.lang.Runnable`

     has only one method: "`void run()`"

  2. Interface

     `java.util.concurrent.Callable`

     has only one method:

     "`V call() throws Exception`"

- `Callable` returns result & may throw exception

- `Runnable` does neither

# How to Execute a Task as a Runnable

Reminder: to execute a `Runnable` ...

```
public class RunnableExample implements Runnable {
    public void run() {
        System.out.println("I am a new thread!");
    }
}

> RunnableExample r = new RunnableExample()
> Thread t = new Thread(r)
> t.start()
I am a new thread!
```

# How to Execute a Task as a Callable

- Can NOT pass a `Callable` to `Thread` constructor

- Instead do this:

```
public class CallableExample implements Callable<String> {
    public String call() {
        return "all done!";
    }
}

> CallableExample<String> c = new CallableExample<String>()
> FutureTask<String> ft = new FutureTask<String>(c)
> Thread t = new Thread(ft) // FutureTask implements Runnable
> t.start()
```

# Future Interface

- `Future<V>` represents a task's eventual result

- Class `FutureTask` implements `Future<V>` interface

- Type of result given by type parameter "`V`"

- Get result by calling `Future`'s `get` method:

```
> CallableExample<String> c = new CallableExample<String>()
> FutureTask<String> ft = new FutureTask<String>(c)
> Thread t = new Thread(ft)
> t.start()
> t.join()    // optional
> String s = ft.get()
> s
all done!
```

# Argument Passing

Q: How to pass arguments to `Runnable` or `Callable`?

A: Pass arguments to *constructor* when object is instantiated

# Argument Passing Example

```
class HashOperation implements Runnable {
    private StringHash ht;
    private String op;
    private String arg;

    public HashOperation(StringHash hashTable,
                         String operation,
                         String argument) {
        this.ht = hashTable;
        this.op = operation;
        this.arg = argument;
    }

    public void run() {
        // logic for HashOperation
    }
}

// pass args via constructor
HashOperation op = new HashOperation(...);
Thread t = new Thread(op);
t.start();
```

# Task Summary, I

If task WILL NOT return a result:

- Write class that implements `Runnable`
- Instantiate object of that class

  ... and call this object's `run()` method to start task

# Task Summary, II

If task WILL return a result:

- Write class that implements `Callable<V>`
- Instantiate object of that class

  ... and call this object's `call()` method to start task
- Instantiate ANOTHER object of some class that implements `Future<V>`

  ... and call this second object's `get()` method to get result (which is of type "`V`")

# Task Summary, III

In either case – task is `Callable<V>` or `Runnable` – pass arguments via constructor

# Thread Pools

- To start a single thread:

```
// r implements Runnable
> Thread t = new Thread(r)
> t.start()
```

- Java 5.0 added classes/methods for "thread pools"

- In particular:

```
ExecutorService pool = new newCachedThreadPool()

ExecutorService pool = new newFixedThreadPool(int nThreads)
```

# Fixed vs. Cached Thread Pools

- Fixed thread pool:
  - Fixed number of threads
  - When given a task to execute ...
    - if a thread is free: assign task to the thread
    - if no thread is free: wait until a thread becomes free
- Cached thread pool:
  - When given a task to execute ...
    - if a thread is free: assign task to the thread
    - if no thread is free: create new thread & assign task to it
  - Cached threads discarded after 60 seconds of disuse

 

# How to Assign Task to Pool Thread

Use `submit` method:

```
int numThreads = 5;
ExecutorService pool = new newFixedThreadPool(numThreads);

//  create task ...
Callable<String> c = new Callable<String>(...);
//  assign task to thread in pool ...
Future<String> f = pool.submit(c);
//  get task's result ...
String s = f.get();
```

# ExecutorService.submit

- One `submit` method takes a `Callable` argument

- Another `submit` method takes a `Runnable` argument

# Task Management

- Task management means:

  - Perform tasks in certain order
  - Perform certain number of tasks at a time
  - Which tasks to shed if system becomes overloaded?

... and so on

- Management is *policy*

- Java *mechanism* for task management:

  - Interface
    `java.util.concurrent.Executor`
    contains only one method:
    "`void execute(Runnable)`"

# Executor Interface

- `execute` method invokes `run` method of its `Runnable` argument at some moment in the future

- Implement `Executor` with some class in order to specialize when & how `Runnable` executes

- One such implementing class: `ThreadPoolExecutor`

- Also: `ExecutorService` is a sub-interface

# ExecutorService Interface

`shutdown` method – refuses to accept more new tasks

`awaitTermination` method – waits for all running threads to terminate

# ThreadPoolExecutor Class

- Class `ThreadPoolExecutor` implements interface `ExecutorService` (`ExecutorService` is sub-interface of `Executor`)

- `ThreadPoolExecutor` gives many options for managing thread pool:

  - Cap on number of threads in cached pool
  - Pre-create certain number of threads in cached pool
  - Timeout different from 60 seconds
  - Detailed control over task assignment & threads; e.g., changing thread priority
  - etc.

# ThreadPoolExecutor: Introduction

- Maintains queue of `Runnables`
- Maintains thread pool

1. Removes `Runnable` object from queue
2. Removes thread from pool
3. Thread executes `run` method of `Runnable`
4. Thread returns to pool when `run` completes

# ThreadPoolExecutor: Queue of Runnables

- May be any class that implements interface `BlockingQueue`

- These are: `ArrayBlockingQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `PriorityBlockingQueue`

# ThreadPoolExecutor: Thread Pool

- Threads created by "thread factory" object

- `Executors` class includes static utility methods for `Executor` interface

- Some of these methods create `ThreadPoolExecutor` object with associated thread factory: `newFixedThreadPool`, `newCachedThreadPool`, `newScheduledThreadPool`

- There is also: `defaultThreadFactory`

# ThreadPoolExecutor: Simple Example

```
// NOTE: uses fixed thread pool
ThreadPoolExecutor exec =
    Executors.newFixedThreadPool(5);


// NOTE: servePage is Runnable object
//       whose run method serves one web page
    for (... each web GET request ...) {
        servePage = ... new object ...
        exec.execute(servePage)
    }
```

# ThreadPoolExecutor: Complex Example

```
// NOTE: provides many specialized arguments,
//       uses default ThreadFactory
ThreadPoolExecutor exec =
    new ThreadPoolExecutor(... thread pool args ...,
                           ... time args ...,
                       new LinkedBlockingQueue<Runnable>());


// NOTE: this part same as previous example
    for (... each web GET request ...) {
        servePage = ... new object ...
        exec.execute(servePage)
    }
```

# Aside: Another Java Convention

Notice correspondence between interface & its class of static utility methods:

- `Collection` interface, `Collections` utility class
- `Executor` interface, `Executors` utility class