Intermediate Java

Daniel Hinojosa

Collection interface

Collections

- Before Java 2, all we had were arrays
- Java 2, introduced `java.util.Collection` package
- Java 5, generics were added to make it easier to use with tools

`List`

- Store elements by insertion order
- 0-based index
- Primitives are boxed

`LinkedList`

- A `List` that is composed of a doubly linked list.
- Constant O(1) time adding and removing elements
- Linear O(n) time for other operations
- Not thread safe

`ArrayList`

- Array's size will be automatically expanded
- Constant Time O(1) for the following
    - `size`
    - `isEmpty`
    - `get` and `set`
    - `iterator` and `listIterator`
- Linear O(n) for all other operations
- Not thread safe

`Set`

- No duplicate elements
- Mathematical `Set` meaning there are more mathematical style methods depending on implementation
- A correct `hashCode` and `equals` must be establish on objects added to any `Set`
- Mutable objects should remain consistent or have an expected behavior

- Prefer immutable objects to avoid unexpected behavior

HashSet

- `Set` backed up by a Hashtable
- No order
- Constant Time O(n) for `add`, `remove`, `contains`, `size`, if `hashCode` is implemented well.
- Iteration speed is proportional to the size
- Not thread safe

TreeSet

- `Set` implements of a `TreeMap`
- Elements are ordered with natural ordering or using a specified `Comparator`
- Made consistent using the `equals` implementation of the contained objects
- Consistent with `equals` requires that `compare` should reflect equality
- All elements are compared using `Comparator` implementation if provided

Map

- `Object` that maps key to a value
- Some have specific order, others do not, depending on
- Some implementations will have restrictions on the types of keys or values
- Mutable objects should remain consistent or have an expected behavior
- Prefer immutable objects to avoid unexpected behavior

TreeMap

- An implementation of `Map`
- Sorted according to the natural ordering of its keys or a given `Comparator`
- O(log(n)) time for `containsKey`, `get`, `put`, `remove` methods
- If a `Comparator` is not provide, the objects contained must correctly implement `equals`

HashMap

- Hash table implementation of `Map`
- Permits `null` keys and values
- Not thread safe
- Constant time for `get` and `put`
- Iteration is time proportional to the capacity

- Determined by two parameters:
    - `initial capacity`: number of buckets
    - `load factory`: how full does the hash table need to before automatically increased
- Rebuilt when entries is greater than the product of load factor and capacity

`Iterator`, `Iterable`, and `Enumeration`

Using `Iterator`

Interface that allows iteration in one direction, forward:

- `hasNext`
- `next`

Using `Iterable`

- `interface` that allows an object to be accepted as way to be included in a `for-each` loop.

Before Java 5:

```
for (Iterator i = suits.iterator(); i.hasNext(); ) {
    Suit suit = (Suit) i.next();
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(suit, j.next()));
}
```

After Java 5:

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

From: https://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html

Using `ListIterator`

Interface that allows iteration in either direction and include calls for:

- `hasPrevious`
- `previous`

Enumeration

- Older way to iterate through collections.
- Has been since less preferred in favor of `Iterator` and `Iterable`

```
for (Enumeration<E> e = v.elements(); e.hasMoreElements();)
    System.out.println(e.nextElement());
```

Source: https://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html

Queue and Deque

Queue

- A collection designed for holding elements prior to processing.
- Used extensively for asynchronous processing in `java.util.concurrent` package
- Typically FIFO (first in, first out), some implementations may be different.
- In FIFO queues, elements are placed at the end or tail
- Queues will have different sorting algorithms

Queue Operations

	Throws Exception	Returns Value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queue Addition Operations

	Throws Exception	Returns Value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- `offer(e)` will add the element typically at the tail of the `Queue`
- `add(e)` will add the element typically at the tail

Queue Removal Operations

	Throws Exception	Returns Value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- `poll` will offer the head element or `null` if empty
- `remove` will offer the head element or throw a `NoSuchElementException`

`Queue` Examination Operations

	Throws Exception	Returns Value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- `element` will retrieve but not remove the head, throws `NoSuchElementException` if empty
- `peek` will retrieve but not remove the head, returns `null` if empty.

`LinkedList` as a `Queue`

```
Queue<Integer> queue = new LinkedList<Integer>();
queue.add(40);
boolean result = queue.offer(50);
assert(result);
boolean result2 = queue.offer(60);
assert(result2);
assert(queue.peek() == 40);
assert(queue.poll() == 40);
```

`PriorityQueue`

- Queue lined up based on *natural ordering* or provided `Comparator`.
- Disallows non-comparable objects
- The *head* element is the least element
- Ties are broken arbitrarily
- Unbounded, with a internal array that is automatically managed
- Not thread-safe
- O(log(n)) for `offer`, `remove`, `poll`, `add`
- O(n) linear for `remove` and `contains`

`PriorityQueue`

Given:

```java
public static class Person {
    private String firstName;
    private String lastName;

    Person(String firstName, String lastName) {..}

    public String getFirstName() {..}

    String getLastName() {..}
}
```

PriorityQueue

Given:

```java
public static class PersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getLastName().compareTo(o2.getLastName());
    }
}
```

PriorityQueue

Using a `PriorityQueue`:

```java
Queue<Person> queue = new PriorityQueue<>(new PersonComparator());
queue.offer(new Person("Franz", "Kafka"));
queue.offer(new Person("Jane", "Austen"));
queue.offer(new Person("Leo", "Tolstoy"));
queue.offer(new Person("Lewis", "Carroll"));
assert(queue.peek().getLastName().equals("Austen"));
```

Deque

- Pronounced *deck*
- Double Ended Queue, allows insertion and removal of elements at both end points
- Implements both `Stack` and `Queue` at the same time

`Stack`

- Old collection from Java 1.x that represents a last in first out collection (LIFO)
- Extended the older `Vector` implementation and provided methods that can be treated as a `Stack`
- Preferable to use `Deque` for stack based operations

`Deque` Operations

Deque Methods

Type of Operation	**First Element (Beginning of the `Deque` instance)**	**Last Element (End of the `Deque` instance)**
Insert	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Remove	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Examine	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

Some extra methods of note: `removeFirstOccurence` removes the first occurrence of the specified element if it exists in the `Deque` instance otherwise remains unchanged.

`removeLastOccurence` removes the last occurrence of the specified element in the `Deque` instance. The return type of these methods is `boolean`, and they return `true` if the element exists in the `Deque` instance.

Lab: Using `Deque`

Threads

Threads

- An independent path of execution with code.
- Multiple threads executing within the same program is a *multithreaded application*
- All Threaded code is performed using `java.lang.Thread`
- In every Java application there is a non-daemon (non-background thread)
- All threads will be executed until:
  - `Runtime.exit()` has been called
  - All non-daemon threads have been terminated

Creating a Basic `Thread`

- Two different philosophies
  - extending `Thread`
  - using a `Runnable` and plugging it into a `Thread`

Extending `Thread`

```java
class MyThread extends Thread {
   private boolean done = false;

   public void finish() {
      this.done = true;
   }

   public void run() {
       while (!done) {
         try {
           Thread.sleep(1000);
         } catch (InterruptedException ie) {
           //ignore
         }
         System.out.print(String.format("In Run: [%s] %s\r\n",
             Thread.currentThread().getName(), LocalDateTime.now()));
       }
   }
}
```

Threads with `Runnable`

- A Thread can be created with instances of the `Runnable` interface
- `Runnable` interface has a `run` method and what is used in the interface is what is run.
- Perfect to have plug the same behavior into multiple `Thread`

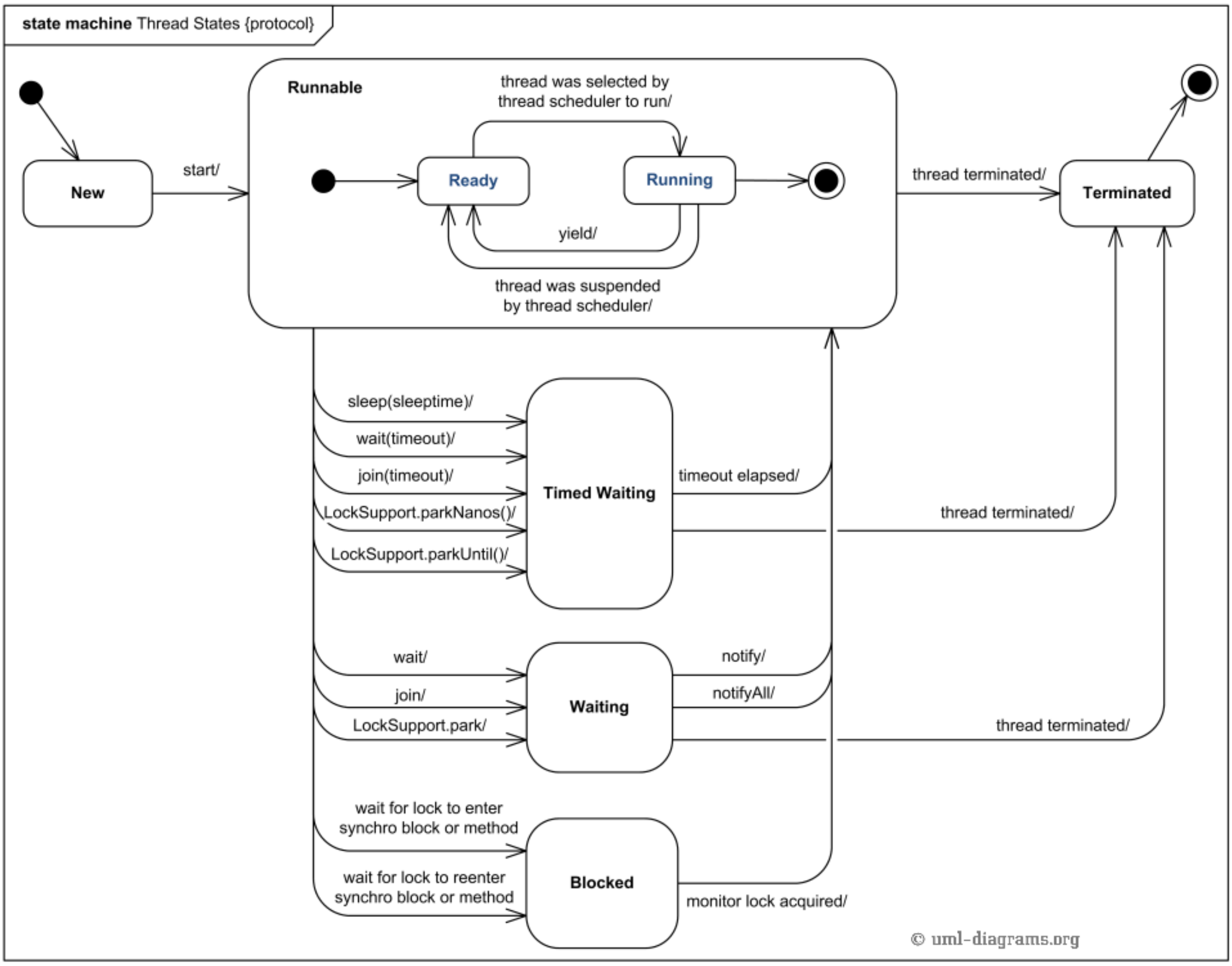
Lab: Create a `Thread` with `Runnable`

```
class MyRunnable implements Runnable {
   private boolean done = false;

   public void finish() {
      this.done = true;
   }

   public void run() {
       while (!done) {
         try {
           Thread.sleep(1000);
         } catch (InterruptedException ie) {
           //ignore
         }
         System.out.print(String.format("In Run: [%s] %s\r\n",
             Thread.currentThread().getName(), LocalDateTime.now()));
       }
   }
}
```

Common `Thread` methods

- `void interrupt()` sends an interrupt signal to a `Thread`
- `static boolean interrupted()` tests if the current `Thread` is interrupted
- `isInterrupted` tests whether a `Thread` is interrupted
- `currentThread` retrieves the current `Thread` in the current scope

Thread states



Thread priorities

- Each thread have a priority.
- Priorities are represented by a number between `1` and `10`.
- Thread Schedulers schedules the threads according to their priority (known as preemptive scheduling).
- Indeterminate because it depends on JVM specification that which scheduling it chooses.
- Predefined constants are available:
  - `MIN_PRIORITY`
  - `MAX_PRIORITY`
  - `NORM_PRIORITY`

`join`

Join allows one thread to wait for another thread to complete. If Thread `t` is running, then the following will cause the current running `Thread` to wait until `t` is done.

```
t.join() //Wait for Thread t to finish and block
```

Lab: `join` Threads

Step 1: In the `ThreadsTest.java` file and in the `com.xyzcorp` package, add the test `testThreadJoin` with the following

```
@Test
public void testThreadJoin() throws InterruptedException {
    Thread thread1 = new Thread() {
        @Override
        public void run() {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.format("Did two seconds on Thread %s\n", Thread.
currentThread().getName());
        }
    };

    thread1.start();
    thread1.join();
    System.out.println("Thread test done");
}
```

Step 2: Run the test

Step 3: Verify the behavior of a join

Daemon Threads

- A daemon thread is a thread that doesn't prevent the JVM from exiting when the thread finishes
- An example of a daemon thread is the garbage collection thread
- Use `setDaemon` to set the `Thread` to a daemon `Thread`.

Lab: Daemon Threads

Step 1: A little different kind of lab, create a class called `DaemonRunner.java` in the `src/main/java` folder:

Step 2: Ensure that it has the following content:

```java
public class DaemonRunner {
    public static void main(String[] args) {
        Thread t = new Thread() {
            @Override
            public void run() {
                while(true) {
                    try {
                        Thread.sleep(2000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("Going...");

                }
            }
        };
        //t.setDaemon(true); //Run first then uncomment
        t.start();
    }
}
```

Step 3: Run and notice that this will continue running until the application is forced to terminate.

Step 4: Uncomment the line `//t.setDaemon(true)` and run again then notice the difference

Immutability

- Immutability is not having the capability of changing an object
- Any change to an object provides a copy

```java
public class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    //equals, hashCode, toString
```

- Processor caching does not need to exchange state.
- Desirable in modern applications.

Race Conditions

- A race condition occurs when two threads or more race to a resource and at the time it is in undesired state.

An inappropriate Singleton

```java
public class MySingleton {
   private static MySingleton instance = null;
   private MySingleton() { }
   public static MySingleton getInstance() {
      if(instance == null) { //What happens when two threads attack this?
         instance = new MySingleton();
      }
      return instance;
   }
}
```

Locks

Intrinsic Locks

- An intrinsic lock is a lock that is innate within the language and provided depending where it is used
- Often called a "monitor lock"
- Intrinsic locks can either be established on a method using the `synchronized` keyword on the method
- Intrinsic locks can also be established on a your selected object
- All threads must establish an "intrinsic lock" on the object.
- Constructors cannot be synchronized since one thread creates objects

Intrinsic Lock on a Method

- The following example shows an intrinsic lock that locks on the `Account` instance that is created

```
class Account {
  private int amount;

  public synchronized void deposit(int amount) {
     this.amount = amount;
  }
}
```

Intrinsic Lock on `this`

```
class Account {
  private int amount;

  public void deposit(int amount) {
     synchronized(this) { // Synchronized on the account object
        this.amount = amount;
     }
  }
}
```

Intrinsic Lock on an external object

```
class Account {
  private Object lock;
  private int amount;
  public Account(Object lock) {
     this.lock = lock;
  }
  public void deposit(int amount) {
     synchronized(lock) { // Synchronized on the account object
        this.amount = amount;
     }
  }
}
```

Intrinsic Lock on a `class`

```
class Account {
  private Object lock;
  private int amount;
  public Account(Object lock) {
     this.lock = lock;
  }
  //The static makes the class become the lock
  public static synchronized void deposit(int amount) {
     this.amount = amount;
  }
}
```

`wait`, `notify`, `notifyAll`

- `wait()` - Causes the current thread to block in the given object until awakened by a `notify()` or `notifyAll()`.
- `notify()`
  - Causes a randomly selected thread waiting on this object to be awakened.
  - It must then try to regain the intrinsic lock.
  - If the “wrong” thread is awakened, your program can deadlock.
- notifyAll()
  - Causes all threads waiting on the object to be awakened
  - Each will then try to regain the monitor lock. Hopefully one will succeed.

Lab: `ResourceThrottle`

Step 1: Create a class called `ResourceThrottle` in `src/main/java` with the following content:

```
package com.xyzcorp;

public class ResourceThrottle {
    private int resourcecount = 0;
    private int resourcemax = 1;

    public ResourceThrottle (int max) {
        resourcecount = 0;
        resourcemax = max;
    }

    public synchronized void getResource (int numberof) {
        while (true) {
            if ((resourcecount + numberof) <= resourcemax) {
                resourcecount += numberof;
                break;
            }
            try {
                wait();
            } catch (Exception e) {}
        }
    }

    public synchronized void freeResource (int numberof) {
        resourcecount -= numberof;
        notifyAll();
    }
}
```

Step 2: Describe the contents of `ResourceThrottle`

Step 3: Create a test in `src/test/java` called `ResourceThrottleTest` that exercises the example.

Volatile Fields

- Volatile files are a flag that the memory is to be read on main memory and not the CPU cache
- If each processor is in charge of it's piece of memory per object they would need to synchronize that state.
- Adding `volatile` to the member variable will avoid "visibility issues"

`volatile` field first guarantee

- If Thread-1 writes to a volatile variable and Thread-2 reads the same variable, all variables visible to Thread-1 before writing the `volatile` variable will flushed to main memory will be visible to the Thread-2
- Reading or Writing by the JVM cannot be rerordered, whatever instructions are meant to happen after the write.

Atomics

- List of values that can be updated atomically.
- Lock-free
- Thread-safe
- Extends the notion of a `volatile` values, fields, and array elements
- All contain the update form of:

```
boolean compareAndSet(expectedValue, updateValue);
```

Atomic Values, Arrays, and Fields

- List of atomics values include:
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicIntegerArray`
  - `AtomicIntegerFieldUpdater`
  - `AtomicLong`
  - `AtomicLongArray`
  - `AtomicLongFieldUpdater`

Atomic References

- `AtomicMarkableReference<V>`
- `AtomicReference<V>`
- `AtomicReferenceArray<E>`
- `AtomicReferenceFieldUpdater<T,V>`
- `AtomicStampedReference<V>`

Without Atomic Variables

Instead of the following `Counter` that is `synchronized` we can opt for an Atomic variable as seen in the next slide.

```
class Counter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

With Atomic Variables

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }
}
```

Deadlocks

- Two or more threads are blocked forever without resolution
- Each thread is waiting on a lock but the other thread has a lock

`Alphonse` and `Gaston` Example

From: https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

```java
class Friend {
  private final String name;
  public Friend(String name) {
      this.name = name;
  }
  public String getName() {
      return this.name;
  }
  public synchronized void bow(Friend bower) {
      System.out.format("%s: %s"
          + "  has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }
  public synchronized void bowBack(Friend bower) {
   System.out.format("%s: %s"
             + " has bowed back to me!%n",
             this.name, bower.getName());
   }
}
```

`Alphonse` and `Gaston` held up

```java
public class DeadlockRunner {
 public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Livelock

- Livelock occurs when two threads are expecting a state from each other but never make it.
- Thread-1 acts as a response to action of Thread-2
- Thread 2 acts as a response to action of Thread-1

The `Criminal` and `Police`

- The `Criminal` demands payment to release the hostage
- The `Police` is waiting for the `Criminal` to release the hostage to receive payment

First the `Criminal`

```java
public class Criminal {
    private boolean hostageReleased = false;

    public void releaseHostage(Police police) {
        while (!police.isMoneySent()) {

            System.out.println(
              "Criminal: waiting police to give ransom");

            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }

        System.out.println("Criminal: released hostage");

        this.hostageReleased = true;
    }

    public boolean isHostageReleased() {
        return this.hostageReleased;
    }
}
```

Then the `Police`

```java
public class Police {
    private boolean moneySent = false;

    public void giveRansom(Criminal criminal) {
        while (!criminal.isHostageReleased()) {
            System.out.println(
               "Police: waiting criminal to release hostage");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }

        System.out.println("Police: sent money");
        this.moneySent = true;
    }

    public boolean isMoneySent() {
        return this.moneySent;
    }
}
```

Running the Livelock

```java
static final Police police = new Police();
static final Criminal criminal = new Criminal();

Thread t1 = new Thread(new Runnable() {
    public void run() {
        police.giveRansom(criminal);
    }
});
t1.start();

Thread t2 = new Thread(new Runnable() {
    public void run() {
        criminal.releaseHostage(police);
    }
});
t2.start();
```

From: http://www.codejava.net/java-core/concurrency/understanding-deadlock-livelock-and-starvation-with-code-examples-in-java

Starvation

- When one greedy thread takes on a resource and doesn't relinquish control
- Either occurs because:
  - One `Thread` priority is higher and will never let go of a resource
  - A `Thread` doesn't finish the job

Starvation by never finishing the job

```
import java.io.*;

public class Worker {

    public synchronized void work() {
        String name = Thread.currentThread().getName();
        String fileName = name + ".txt";

        try (
            BufferedWriter writer =
               new BufferedWriter(new FileWriter(fileName));
        ) {
            writer.write("Thread " + name + " wrote this mesasge");
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        while (true) { //Keep going and never let go
            System.out.println(name + " is working");
        }
    }
}
```

Java 5 Concurrent Features

Reentrant Locks

- Same semantics as an implicit monitor lock accessed by `synchronized`
- The `RentrantLock` is owned by the thread last successfully locking, but not unlocking
- May contain a `fairness` operator, when `true`, favors longer waiting threads
- Standard practice to use a `try/catch` block to access the lock and unlock

```java
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...

   public void m() {
     lock.lock();  // block until condition holds
     try {
       // ... method body
     } finally {
       lock.unlock()
     }
   }
 }
```

This lock supports a maximum of 2147483647 recursive locks by the same thread

From: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html

Thread safe collections

- `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.`
- `ConcurrentMap` is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`.
- `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches. The standard general-purpose implementation of `ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`.

Futures

> Future def. - Future represents the lifecycle of a task and provides methods to test whether the task has completed or has been cancelled.

> Future can only move forwards and once complete it stays in that state forever.

Thread Pools

Before setting up a future, a thread pool is required to perform an asynchronous computation. Each pool with return an `ExecutorService`.

There are a few thread pools to choose from:

- `FixedThreadPool`
- `CachedThreadPool`
- `SingleThreadExecutor`
- `ScheduledThreadPool`
- `ForkJoinThreadPool`

Fixed Thread Pool

- "Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue."
- Keeps threads constant and uses the queue to manage tasks waiting to be run
- If a thread fails, a new one is created in its stead
- If all threads are taken up, it will wait on an unbounded queue for the next available thread

Cached Thread Pool

- Flexible thread pool implementation that will reuse previously constructed threads if they are available
- If no existing thread is available, a new thread is created and added to the pool
- Threads that have not been used for sixty seconds are terminated and removed from the cache

Single Thread Executor

- Creates an Executor that uses a single worker thread operating off an unbounded queue
- If a thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

Scheduled Thread Pool

- Can run your tasks after a delay or periodically
- This method does not return an `ExecutorService`, but a `ScheduledExecutorService`
- Runs periodically until `canceled()` is called.

Fork Join Thread Pool

- An `ExecutorService`, that participates in *work-stealing*
- By default when a task creates other tasks (`ForkJoinTasks`) they are placed on the same on queue as the main task.
- *Work-stealing* is when a processor runs out of work, it looks at the queues of other processors and "steals" their work items.
- Not a member of Executors. Created by instantiation
- Brought up since this will be in many cases the "default" thread pool on the JVM

Basic Future Blocking (JDK 5)

```
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);

Callable<Integer> callable = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        System.out.println("Inside ze future: " +
                Thread.currentThread().getName());
        System.out.println("Future priority: " + Thread.currentThread().
getPriority());
        Thread.sleep(5000);
        return 5 + 3;
    }
};

System.out.println("In test:" + Thread.currentThread().getName());
System.out.println("Main priority" + Thread.currentThread().getPriority());
Future<Integer> future = fixedThreadPool.submit(callable);

//This will block
Integer result = future.get(); //block
System.out.println("result = " + result);
```

Basic Future Asynchronous (JDK 5)

```java
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

Callable<Integer> callable = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        Thread.sleep(3000);
        return 5 + 3;
    }
};

Future<Integer> future = cachedThreadPool.submit(callable);

//This is proper asynchrony
while (!future.isDone()) {
    System.out.println("I am doing something else on thread: " +
            Thread.currentThread().getName());
}

Integer result = future.get();
```

Futures with Parameters

- `Future` with a parameter will require a parameter be made with method and use a `final` variable for the future

Lab: Creating a Future with a Parameter

Step 1: Create the following test in the `src/test/java` folder in the `FuturesTest.java` file

```java
private Future<Stream<String>> downloadingContentFromURL(final String url) {
    ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
    return cachedThreadPool.submit(new Callable<Stream<String>>() {
        @Override
        public Stream<String> call() throws Exception {
            URL netUrl = new URL(url);
            URLConnection urlConnection = netUrl.openConnection();
            BufferedReader reader = new BufferedReader(
                    new InputStreamReader(
                            urlConnection.getInputStream()));
            return reader
                    .lines()
                    .flatMap(x -> Arrays.stream(x.split(" ")));
        }
    });
}
```

Step 2: Ensure it compiles, and explain what is possibly happening.

Lab : Test the Future with a parameter

Step 1: In src/test/java in the FuturesTest.java file create testGettingURL with the following content:

```
@Test
public void testGettingUrl() throws ExecutionException, InterruptedException {
    Future<Stream<String>> future = downloadingContentFromURL
            (<FILL_IN_WEBSITE>);
    while (!future.isDone()) {
        Thread.sleep(1000);
        System.out.println("Doing Something Else");
    }
    Stream<String> allStrings = future.get();
    allStrings
            .filter(x -> x.contains("Ohio"))
            .forEach(System.out::println);
    Thread.sleep(5000);
}
```

Completable Future

- Staged Completions of Interface java.util.concurrent.CompletionStage<T>
- Ability to chain functions to Future<V>
- Analogies
  - thenApply(…) = map
  - thenCompose(…) = flatMap
  - thenCombine(…) = independent combination
  - thenAccept(…) = final processing

Lab: Setting up the CompletableFuture

Step 1: Setup the following member variables in FuturesTest

```
private CompletableFuture<Integer> integerFuture1;
private CompletableFuture<Integer> integerFuture2;
private CompletableFuture<String> stringFuture1;
private ExecutorService executorService;
```

Lab: Create A Thread Pool and an asynchronous `CompletableFuture`

Step 1: In a method called `setUp` and annotated with `@Before` establish an `ExecutorService` and the first `CompletableFutures`

```
@Before
public void setUp() {
   executorService = Executors.newCachedThreadPool();


  integerFuture1 = CompletableFuture
          .supplyAsync(new Supplier<Integer>() {
              @Override
              public Integer get() {
                  try {
                      System.out.println("intFuture1 is Sleeping in thread: "
                              + Thread.currentThread().getName());
                      Thread.sleep(3000);
                  } catch (InterruptedException e) {
                      e.printStackTrace();
                  }
                  return 5;
              }
          });
}
```

Lab: Create two more asynchronous `CompletableFuture`

Step 1: In a method called `setUp` and annotated with `@Before` establish two more `CompletableFuture`

```java
@Before
public void setUp() {

  ...

  integerFuture2 = CompletableFuture
          .supplyAsync(() -> {
              try {
                  System.out.println("intFuture2 is sleeping in thread: "
                          + Thread.currentThread().getName());
                  Thread.sleep(400);
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }
              return 555;
          }, executorService);

  stringFuture1 = CompletableFuture
          .supplyAsync(() -> {
              try {
                  System.out.println("stringFuture1 is sleeping in thread: "
                          + Thread.currentThread().getName());
                  Thread.sleep(4300);
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }
              return "Los Angeles, CA";
          });
}
```

Lab: Using the `CompletableFuture` with `thenAccept`

Step 1: In `FuturesTest` create a test method called `completableFutureWithThenAccept` with the following:

```java
@Test
public void completableFutureWithThenAccept() throws InterruptedException {
    integerFuture1.thenAccept(System.out::println);
    Thread.sleep(5000);
}
```

Step 2: Describe why there is a `sleep` at the end of this method.

Step 3: Run the test

Lab: Using an equivalent `map` with `thenApply`

Step 1: In `FuturesTest` create a test method called `completableFutureWithThenApply` with the following:

```
@Test
public void completableFutureWithThenApply() throws InterruptedException {
    CompletableFuture<String> future =
            integerFuture1.thenApply(x -> {
                System.out.println("In Block:" +
                        Thread.currentThread().getName());
                return "" + (x + 19);
            });
    future.thenAccept(s -> {
        System.out.println(Thread.currentThread().getName());
        System.out.println(s);
    });
    Thread.sleep(5000);
}
```

Step 2: Run the test

Lab: Using an equivalent `map` with `thenApplyAsync`

- `thenApplyAsync` will apply a map but will do so on another `Thread`

Step 1: In `FuturesTest` create a test method called `completableFutureWithThenApplyAsync` with the following:

```java
@Test
public void completableFutureWithThenApplyAsync() throws InterruptedException {
    CompletableFuture<String> thenApplyAsync =
            integerFuture1.thenApplyAsync(x -> {
                System.out.println("In Block:" +
                        Thread.currentThread().getName());
                return "" + (x + 19);
            }, executorService);
    Thread.sleep(5000);

    thenApplyAsync.thenAcceptAsync((x) -> {
        System.out.println("Accepting in:" + Thread.currentThread().getName());
        System.out.println("x = " + x);
    });

    System.out.println("Main:" + Thread.currentThread().getName());
    Thread.sleep(3000);
}
```

Step 2: Run the test

Lab: `thenRun`

- `thenRun` will run any block after the chain of `CompleteableFuture`
- It will return a `CompletableFuture<Void>` so essentially it is sentinel.

Step 1: In `FuturesTest` create a test method called `completableFutureWithThenRun` with the following:

```java
@Test
public void completableFutureWithThenRun() throws InterruptedException {
    integerFuture1.thenRun(new Runnable() {
        @Override
        public void run() {
            String successMessage =
                    "I am doing something else once" +
                            " that future has been triggered!";
            System.out.println
                    (successMessage);
        }
    });
    Thread.sleep(3000);
}
```

Step 2: Run the test

Lab: Trapping Errors with `exceptionally`

- Exceptionally takes an error exception if anywhere on the chain there is an `Exception` thrown

Step 1: In `FuturesTest` create a test method called `completableFutureWithExceptionally` with the following:

```
@Test
public void completableFutureExceptionally() throws InterruptedException {
    stringFuture1.thenApply((s) -> Integer.parseInt(s))
            .exceptionally(t -> {
                //t.printStackTrace();
                return -1;}).thenAccept(System.out::println);
    System.out.println("This message should appear first.");
    Thread.sleep(6000);
}
```

Step 2: Run the test

Lab: Trapping Errors with `handle`

- If you wish to handle the error based on both a successful output or an exception, use `handle`

Step 1: In `FuturesTest` create a test method called `completableFutureWithHandle` with the following:

```
stringFuture1.thenApply((s) -> Integer.parseInt(s)).handle(
        new BiFunction<Integer, Throwable, Integer>() {
            @Override
            public Integer apply(Integer item, Throwable throwable) {
                if (throwable == null) return item;
                else return -1;
            }
        }).thenAccept(System.out::println);

Thread.sleep(6000);
```

Step 2: Run the test

Lab: `flatMap` with `compose`, but first a `ComposableFuture` with a parameter

- Notice the structure is the same as a regular `Future` with a parameter
- We need to encapsulate the future in a method using the parameter

Step 1: Create a method in `FuturesTest` called `getTemperatureInFahrenheit` with the following:

```
public CompletableFuture<Integer>
    getTemperatureInFahrenheit(final String cityState) {
    return CompletableFuture.supplyAsync(() -> {
        //We go into a webservice to find the weather...
        System.out.println("In getTemperatureInFahrenheit: " +
                Thread.currentThread().getName());
        System.out.println("Finding the temperature for " + cityState);
        return 78;
    });
}
```

Step 2: Ensure that there are no errors

Lab: Using `compose`

- `compose` is `flatMap` for `CompletableFuture` and allows you to build off one another

Step 1: Create a test in `FuturesTest` called `completableCompose` with the following:

```
@Test
public void completableCompose() throws InterruptedException {
    CompletableFuture<Integer> composition =
            stringFuture1.thenCompose(s -> getTemperatureInFahrenheit(s));
    composition.thenAccept(System.out::println);
    Thread.sleep(6000);
}
```

Step 2: Run the test

Lab: Using `combine`

- `combine` is not reliant on another's evaluation but is used as a `join` to join the `CompletableFuture`

Step 1: Create a test in `FuturesTest` called `completableCombine` with the following:

```java
@Test
public void completableCombine() throws InterruptedException {
    CompletableFuture<Integer> combine =
            integerFuture1
                    .thenCombine(integerFuture2, (x, y) -> x + y);
    combine.thenAccept(System.out::println);
    Thread.sleep(6000);
}
```

Step 2: Run the test

A Promise is a `Promise`

- A promise is a `Future` that not determined by calculation
- There is no `Promise` construct in Java per se
- You can use a `CompletableFuture` to perform the action of a Promise

Lab: Creating a Promise using `CompletableFuture`

Step 1: Create a test in `FuturesTest` called `testCompletableFuturePromise` with the following:

```java
@Test
public void testCompletableFuturePromise() throws InterruptedException {
    CompletableFuture<Integer> completableFuture =
            new CompletableFuture<>();

    completableFuture.thenAccept(System.out::println);

    System.out.println("Processing something else");
    Thread.sleep(1000);
    completableFuture.complete(42);
    Thread.sleep(3000);
}
```

Step 2: Run the test

Step 3: Discuss the how this is a Promise