Intermediate Java

Daniel Hinojosa

Setup

Lab: Pre-Class Check

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8 (latest java is 1.8.0_131)
- Maven 3.5.0

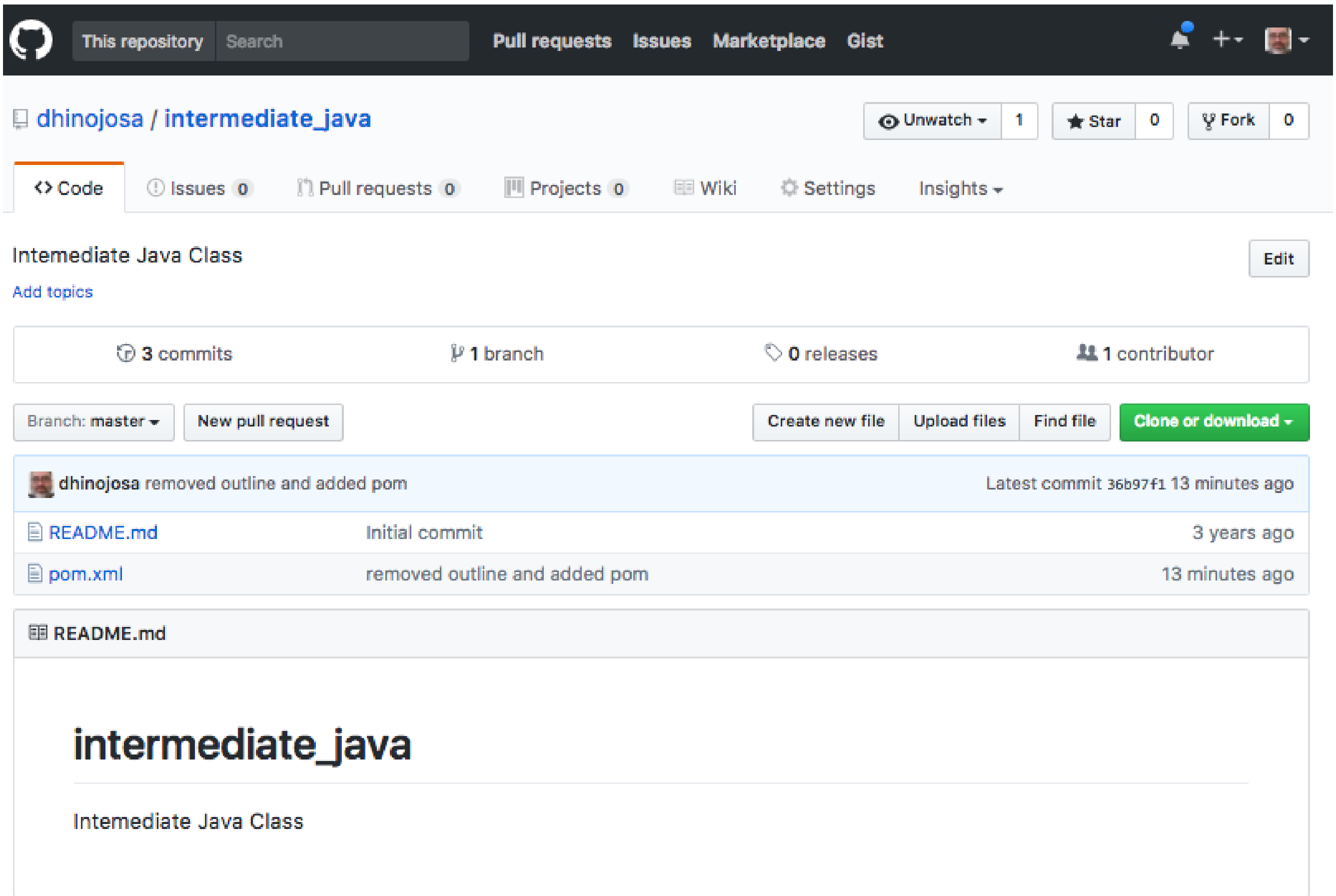
To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_131

% java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_131-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.5.0 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T09:41:47
-07:00)
Maven home: /usr/lib/mvn/apache-maven-3.5.0
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /usr/lib/jvm/jdk1.8.0_131/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-34-generic", arch: "amd64", family: "unix"
```

The JDK 8 Version doesn't have to be exact as long as it is Java 8.

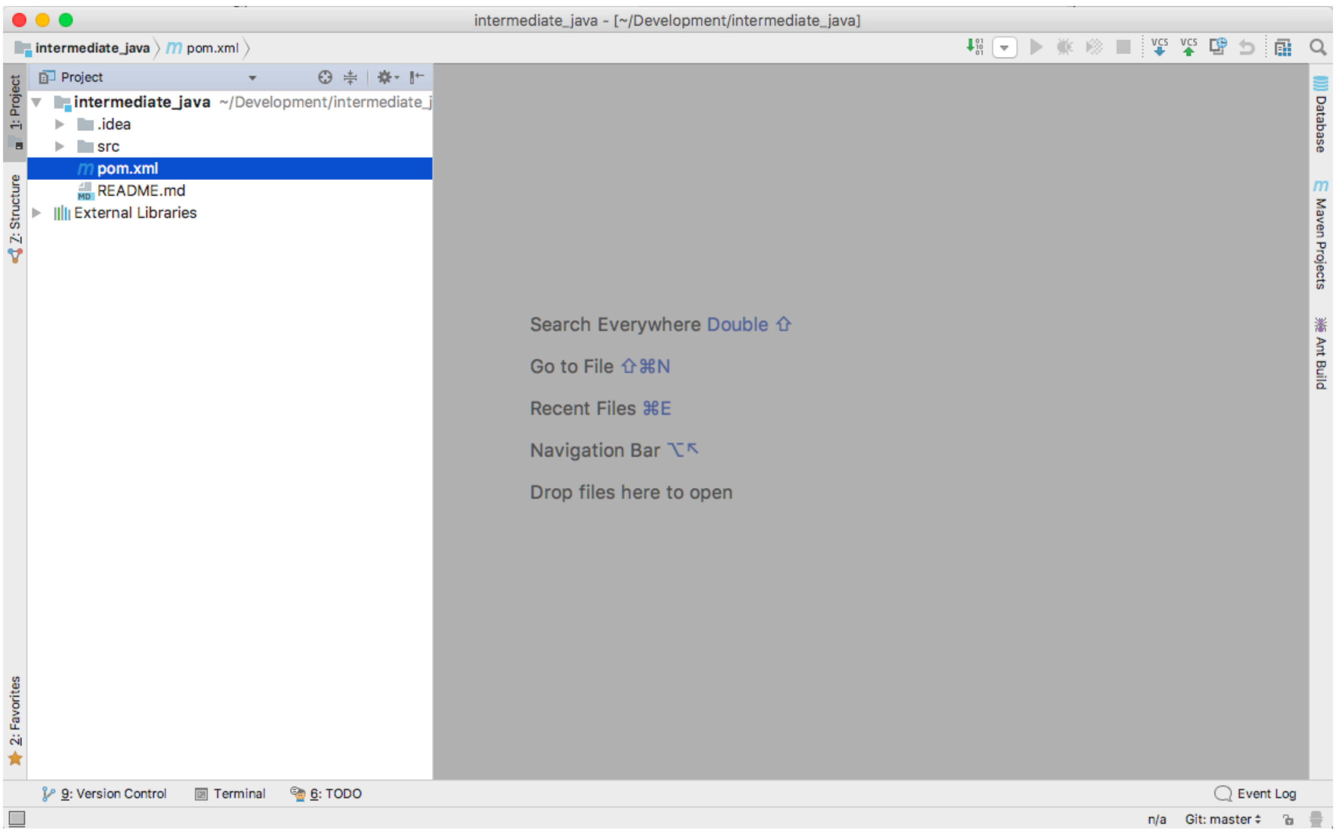
Lab: Download the Project

From https://github.com/dhinojosa/intermediate_java download the project .zip file and extract it into your favorite location or if you know how to use git, then clone the project into your favorite location.



Optional Lab: Open Project in IntelliJ

Once intermediate_java is downloaded and extracted or cloned to your favorite location, In IntelliJ Open The Project, IntelliJ will recognize it as a Maven project and you are good to go.

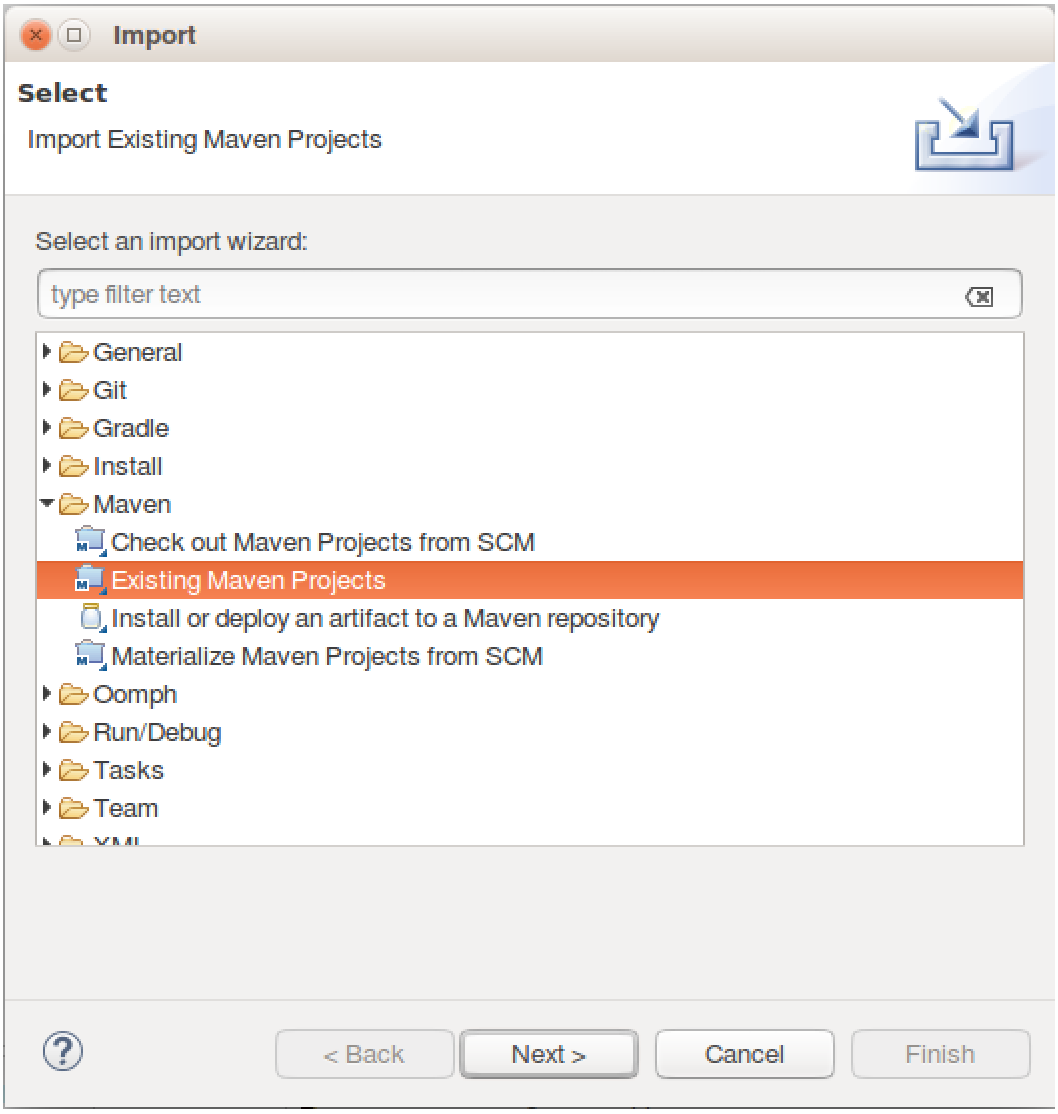


Optional Lab: Open Project in Eclipse

Once downloaded and extracted:

Step 1: Select *File* > *Import Project* in the menu.

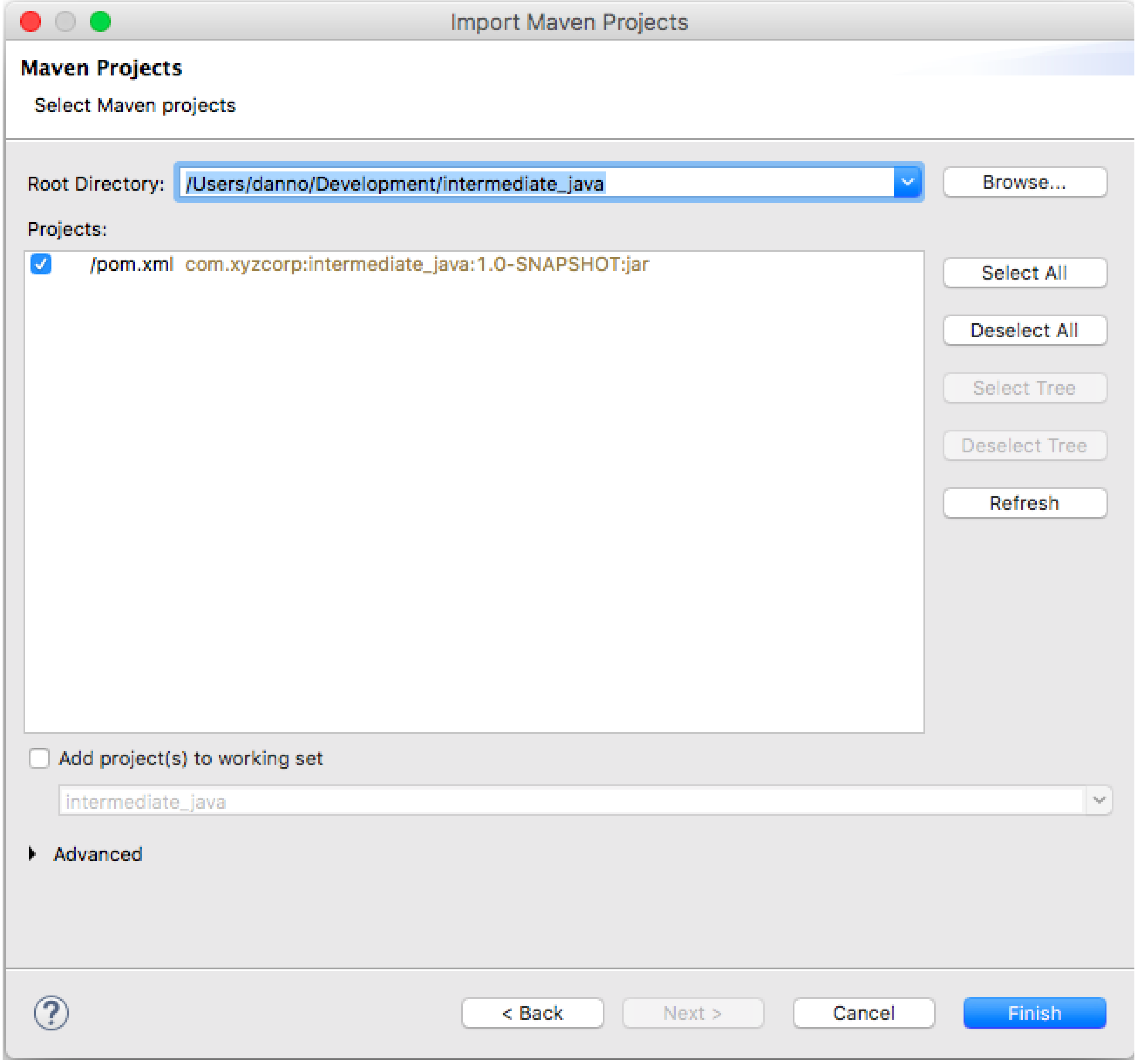
Step 2: In the following dialog box:



- Open the *Maven* category
- Select *Import Existing Maven Projects*

Optional Lab: Open Project in Eclipse (Continued)

Step 3:



- Click the *Browse:* button next to *Root Directory*
- Select the location of your ***intermediate_java*** directory.

Step 4: Click *Finish*

Lambdas

About Java 8 Lambdas

Functional Interface Definition

> A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it.

(`equals` is an explicit declaration of a concrete method inherited from `Object` that, without this declaration, would otherwise be implicitly declared.)

Default Methods

- Enable you to add new functionality to the `interface` of your libraries
- Ensure binary compatibility with code written for older versions of those `interface`.
- Comes closer to have "concrete" method in an "interface" by composing other `abstract` methods.

Default Method Arbitrary Example

```java
public interface Human {
   public String getFirstName();
   public String getLastName();
   default public String getFullName() {
     return String.format("%s %s",
        getFirstName(), getLastName());
   }
}
```

Lab: Create `MyPredicate`

Step 1: Ensure you have a `src/main/java` directory in the `intermediate_java` module

Step 2: Ensure that the folders are seen as a build path (Eclipse only)

Step 3: Create a package called `com.xyzcorp` in `src/main/java`

Step 4: Create an interface in `com.xyzcorp` called `MyPredicate`

```java
package com.xyzcorp;

public interface MyPredicate<T> {
	public boolean test(T item);
}
```

About `MyPredicate`

- It's an interface
- One `abstract` method: `test`
- `default` methods don't count (More on that later)
- `static` methods don't count
- Any methods inherited from `Object` don't count either.

```java
package com.xyzcorp;

public interface MyPredicate<T> {
	public boolean test(T item);
}
```

Conclusion: We can omit the name when we implement it.

Functional `filter`

> Filter is a higher-order function that processes a data structure (usually a list) in some order to produce a new data structure containing exactly those elements of the original data structure for which a given predicate returns the boolean value true.

Wikipedia: Map (higher-order function)

Functional `filter` by example

1. Given List of `list`: `[1,2,3,4]`
2. Given a function `f`: `x → x % 2 == 0`
3. When calling `filter` on a `list` with `f`: `[1,2,3,4].filter(f)`
4. Then a copy of the `list` should return: `[2,4]`

Lab: Using `MyPredicate`

Step 1: Create a File in the `com.xyzcorp` package called ***Functions.java***

Step 2: Create an method called `myFilter` as seen below.

```
package com.xyzcorp;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Functions {

    public static <T> List<T> myFilter (List<T> list, MyPredicate<T> predicate) {
        ArrayList<T> result = new ArrayList<T>();
        for (T item : list) {
            if (predicate.test(item)) {
                result.add(item);
            }
        }
        return result;
    }
```

This is the functional `filter`

Lab: Test Method in *LambdasTest.java*

Step 1: Ensure you have a `src/test/java` directory in the `intermediate_java` module

Step 2: Ensure that the folders are seen as a build path (Eclipse only)

Step 3: Create a package called `com.xyzcorp` in `src/test/java`

Step 4: Create a class called `LambdasTest` in the `com.xyzcorp` package with the following test:

```java
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    @Test
    public void testMyFilter() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15,
                                              19, 21, 33, 78, 93, 10);
        List<Integer> filtered = Functions.myFilter(numbers,
           new MyPredicate<Integer>() {
               @Override
               public boolean test(Integer item) {
                   return item % 2 == 0;
               }
           });
        System.out.println(filtered);
    }
}
```

Here we are defining what the predicate will do when sent into `filter`.

Step 5: Run the test in your IDE to verify that it works as expected

Lab: `MyPredicate` is "Lambdaized"

Step 1: In the test you just wrote, convert `MyPredicate` into a lambda and use your IDE's faculties to do so.

```java
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

   @Test
    public void testMyFilter() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15,
                                              19, 21, 33, 78, 93, 10);
        List<Integer> filtered = Functions.myFilter(numbers, item -> item % 2 == 0);
        System.out.println(filtered);
    }
}
```

Functional `map`

> Applies a given function to each element of a list, returning a list of results in the same order. It is often called apply-to-all when considered in functional form.

Wikipedia: Map (higher-order function)

Functional `map` by example

1. Given List of `list`: `[1,2,3,4]`
2. Given a function `f`: `x → x + 1`
3. When calling `map` on a `list` with `f`: `[1,2,3,4].map(f)`
4. Then a copy of the `list` should return: `[2,3,4,5]`

Lab: Create a `MyFunction`

Step 1: Create an `interface` for `MyFunction`

- In `src/main/java` and in the package `com.xyzcorp` create an `interface` called `MyFunction`
- The interface should have a method called `apply`
- The `MyFunction` interface should have two parameterized types `T1` and `R`
- The `apply` method have one parameter `(T1 in)`
- The `apply` method should have one return type: `R`

Lab: Create a `myMap` in *Functions.java*

Step 1: Create `static` method called `myMap` in ***Functions.java*** with the following method header:

```
public static <T, R> List<R> myMap(List<T> list, MyFunction<T, R> function) { }
```

Step 2: Fill in the method with what you believe a `map` should look like given the previous description.

Lab: Use `myMap` in *LambdasTest.java*

Step 1: Add the following test to your ***LambdasTest.java*** file:

```
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMyMap() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                              21, 33, 78, 93, 10);
        List<Integer> mapped = Functions.myMap(numbers,
          new MyFunction<Integer, Integer>() {
            @Override
            public Integer apply(Integer item) {
                return item + 2;
            }
        });
        System.out.println(mapped);
    }
}
```

Step 2: Convert the `new MyFunction` anonymous instantiation into a lambda using your IDE's faculties

Step 3: Run to verify it all works!

Functional `forEach`

> Performs an action on each element returning nothing or `void`, a sink

Functional `forEach` by example

1. Given List of `list`: `[1,2,3,4]`
2. Given a function `f`: `x -> System.out.println(x)`
3. When calling `forEach` on a `list` with `f`: `[1,2,3,4].forEach(f)`
4. Then `void` is returned. This is called a side effect.

Lab: Create `MyConsumer`

Step 1: Under `src/main/java`, and inside the `com.xyzcorp` package, create an `interface` called `MyConsumer` with the following content:

```
package com.xyzcorp;

public interface MyConsumer<T> {
   public void accept(T item);
}
```

Notice that it does not return anything

Lab: Create a `forEach` in *ListOps.java*

Step 1: Create `static` method called `myForEach` in ***Functions.java*** with the following method header:

```
public static <T, R> void myForEach(List<T> list, MyConsumer<T> consumer) {}
```

Step 2: Fill in the method with what you believe a `forEach` should look like

Lab: Use `myForEach` in *LambdasTest.java*

Step 1: Add the following test to your ***LambdasTest.java*** file:

```java
package com.xyzcorp;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testForEach() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                              21, 33, 78, 93, 10);
        Functions.myForEach(numbers, new MyConsumer<Integer>() {
            @Override
            public void consume(Integer item) {
                System.out.println(item);
            }
        });
    }
}
```

Step 4: Convert the `new MyConsumer` anonymous instantiation into a lambda using your IDE's faculties

Step 5: Run to verify it all works!

A Detour with Method References

- When a lambda expression does nothing but call an existing method
- It's often clearer to refer to the existing method by name.
- Works with lambda expressions for methods that already have a name.

Types of Method References

Table 1. Types of Method References

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Lab: `forEach` with a method reference

Step 1: Convert `x → System.out.println(x)` from the `testForEach` exercise in ***LambdasTest.java*** into a method reference.

```
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testForEach() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                               21, 33, 78, 93, 10);
        Functions.myForEach(numbers, System.out::println);
    }
}
```

Although confusing, in `System.out`, `out` is a `public final static` variable. Therefore, `println` is a non-static method of `java.io.PrintStream`. This is an instance method of an object.

Lab: Method Reference to a static method

Step 1: Enter the following in the test method, `testMethodReferenceAStaticMethod` into ***LambdasTests.java*** and convert it using a method reference.

```
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMethodReferenceAStaticMethod() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                              21, 33, 78, 93, 10);
        System.out.println(Functions.myMap(numbers, a -> Math.abs(a)));
    }
}
```

Use your IDE to guide you. It's easier that way.

Step 2: Run to verify it all works!

Lab: Method Reference with a Containing Type

Step 1: Enter the following test method `testMethodReferenceAContainingType` in ***LambdasTest.java*** and convert it using a method reference.

```
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMethodReferenceAContainingType() {
        List<String> words = Arrays.asList("One", "Two", "Three", "Four");
        System.out.println(Functions.myMap(words, s -> s.length()));
    }
}
```

Use your IDE to guide you. It's easier that way.

Step 2: Run to verify it all works!

Lab: Method Reference with a Containing Type Trick Question

Step 1: Enter the following test method `testMethodReferenceAContainingTypeTrickQuestion` in ***LambdasTest.java*** and convert it using a method reference.

```
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMethodReferenceAContainingTypeTrickQuestion() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                              21, 33, 78, 93, 10);
        System.out.println(Functions.myMap(numbers, number -> number.toString()));
    }
}
```

Use your IDE to guide you. It's easier that way.

Step 2: Run to verify it all works!

Lab: Create a `Tax Rate` class:

Step 1: In `src/main/java`, create a file called ***TaxRate.java*** in the `com.xyzcorp` package with the following content:

```java
package com.xyzcorp;

public class TaxRate {
    private final int year;
    private final double taxRate;

    public TaxRate(int year, double taxRate) {
        this.year = year;
        this.taxRate = taxRate;
    }

    public double apply(int subtotal) {
        return (subtotal * taxRate) + subtotal;
    }
}
```

Step 2: Ensure it compiles.

Lab: Method Reference with an Instance

Step 1: Enter the following test method `testMethodReferenceAnInstance` in ***LambdasTest.java*** and convert it using a method reference.

```java
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMethodReferenceAnInstance() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                              21, 33, 78, 93, 10);
        TaxRate taxRate2016 = new TaxRate(2016, .085);
        System.out.println(Functions.myMap(numbers, subtotal -> taxRate2016.apply
(subtotal)));
    }
}
```

Use your IDE to guide you. It's easier that way.

Step 2: Run to verify it all works!

Lab: Method Reference with an New Type

Step 1: Enter the following test method `testMethodReferenceANewType` in ***LambdasTest.java*** and convert it using a method reference.

```
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMethodReferenceANewType() {
        List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                                              21, 33, 78, 93, 10);
        System.out.println(Functions.myMap(numbers, value -> new Double(value)));
    }
}
```

Use your IDE to guide you. It's easier that way.

Step 2: Run to verify it all works!

Lab: Create `MySupplier`

Step 1: In `src/main/java`, create an `interface` in the `com.xyzcorp` package called `MySupplier`

```
package com.xyzcorp;

public interface MySupplier<T> {
   public T get();
}
```

Compare the difference to `MyConsumer`

Lab: Create a `myGenerate` in *Functions.java*

Step 1: Create `static` method called `myGenerate` with the following method header which takes a `MySupplier`, and a count, and returns a `List` with `count` number of items where each element is derived from invoking the `Supplier`

```java
public static <T> List<T> myGenerate(MySupplier<T> supplier, int count) {}
```

Step 2: Fill in the method with what you believe a `myGenerate` should look like

Lab: Use `myGenerate` in *LambdasTest.java*

Step 1: Add the following test, `testMyGenerate` to the `LambdasTests` class:

```java
package com.xyzcorp;

import org.junit.Test;

import java.time.LocalDateTime;
import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testMyGenerate() {
        List<LocalDateTime> localDateTimes =
          Functions.myGenerate(new MySupplier<LocalDateTime>() {
              @Override
              public LocalDateTime get() {
                  return LocalDateTime.now();
              }
          }, 10);
        System.out.println(localDateTimes);
    }
}
```

`LocalDateTime.now()` is from the new Java Date/Time API from Java 8.

Step 2: Convert the `new MySupplier` anonymous instantiation into a lambda using your IDE's faculties

Step 3: Run to verify it all works!

Lab: Viewing Consumer, Supplier, Predicate, Function, in the official Javadoc.

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Lab: Multi-line Lambdas

Step 1: In ***LambdasTest.java*** create the following test, `testLambdasWithRunnable` where a `java.lang.Runnable` and `java.lang.Thread` is being created.

```
package com.xyzcorp;

import org.junit.Test;

import java.time.LocalDateTime;
import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testLambdasWithRunnable() {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                String threadName =
                        Thread.currentThread().getName();
                System.out.format("%s: %s%n",
                        threadName,
                        "Hello from another thread");
            }
        });
        t.start();
    }
}
```

`Runnable` is an `interface` with one `abstract` method.

Step 2: Convert the `Runnable` into a lambda.

Step 3: Notice how the lambda is created, this is a multi-line lambda.

Closure

- *Lexical scoping* caches values provided in one context for use later in another context.
- If lambda expression closes over the scope of its definition, it is a *closure*.

```java
public static Integer foo
        (Function<Integer, Integer> f) {
    return f.apply(5);
}

public void otherMethod() {
    Integer x = 3;
    Function<Integer, Integer> add3 = z -> x + z;
    System.out.println(foo(add3));
}
```

Lexical Scoping Restrictions

- To avoid any race conditions:
  - The variable that is being in enclosed must either be:
    - `final`
    - *Effectively final*. No change can be made after used in a closure.

Closure Error

The following will not work…

```java
public static Integer foo
        (Function<Integer, Integer> f) {
    return f.apply(5);
}

public void otherMethod() {
    Integer x = 3;
    Function<Integer, Integer> add3 = z -> x + z;
    x = 10;
    System.out.println(foo(add3));
}
```

Lab: Create Duplicated Code

An application for a closure is to avoid repetition.

Step 1: In ***LambdasTest.java*** create the following test, `testClosuresAvoidRepeats`

```java
package com.xyzcorp;

import org.junit.Test;

import java.time.LocalDateTime;
import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    @Test
    public void testClosuresAvoidRepeats() {
        MyPredicate<String> stringHasSizeOf4 =
                str -> str.length() == 4;

        MyPredicate<String> stringHasSizeOf2 =
                str -> str.length() == 2;

        List<String> names = Arrays.asList("Foo", "Ramen", "Naan", "Ravioli");
        System.out.println(Functions.myFilter(names, stringHasSizeOf4));
        System.out.println(Functions.myFilter(names, stringHasSizeOf2));
    }
}
```

Step 2: Notice that `stringHasSize4` and `stringHasSize2` are duplicated.

Lab: Refactor Duplicated Code with a Closure

An application for a closure is to avoid repetition.

Step 1: In ***LambdasTest.java*** change `testClosuresAvoidRepeats` to avoid repeats to look like the following:

```java
package com.xyzcorp;

import org.junit.Test;

import java.time.LocalDateTime;
import java.util.Arrays;
import java.util.List;

public class LambdasTest {

    ...

    public MyPredicate<String> stringHasSizeOf(final int length) {
        return null; //Create your closure here
    }

    @Test
    public void testClosuresAvoidRepeats() {
        List<String> names = Arrays.asList("Foo", "Ramen", "Naan", "Ravioli");
        System.out.println(Functions.myFilter(names, stringHasSizeOf(4)));
        System.out.println(Functions.myFilter(names, stringHasSizeOf(2)));
    }
}
```

Step 2: Inside of `stringHasSizeOf(final int length)` return a `MyPredicate` that *closes* around the length.

`Optional`

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

`Optional` Defined in Java 8

> A **container object** which may or may not contain a non-null value. If a value is present, `isPresent()` will return `true` and `get()` will return the value.

Optional is **not** `Serializable`

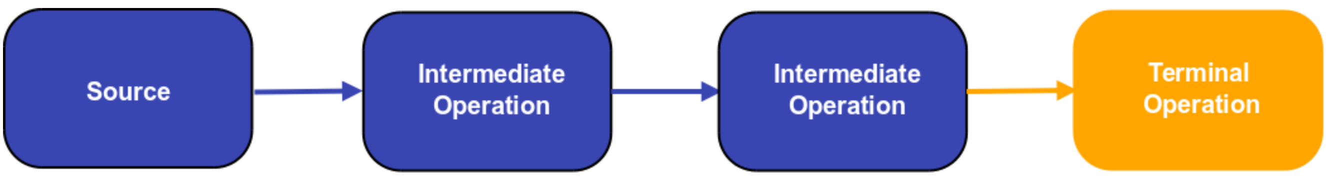
This is a value-based class; use of identity-sensitive operations (including reference equality (`==`), identity hash code, or synchronization) on instances of `Optional` may have unpredictable results and should be avoided.

Streams

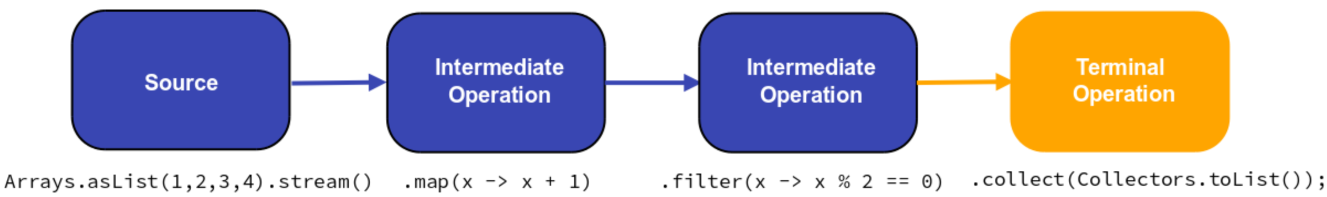
`Streams` differ from Collections in the following ways:

- No storage. A stream is not a data structure that stores elements; instead
- It conveys elements from a source through a pipeline of computational operations
- Sources can include.
  - Data structure
  - An array
  - Generator function
  - I/O channel
- Functional in nature. An operation on a stream produces a result, **but does not modify its source**.
- Intermediate operations are laziness-seeking exposing opportunities for optimization.
- Possibly unbounded. While collections have a finite size, streams need not.
- Short-circuiting operations such as `limit(n)` or `findFirst()` can allow computations on infinite streams to complete in finite time.
- Consumable, The elements of a stream are only visited once during the life of a stream.
- Like an `java.util.Iterator`, a new `Stream` must be generated to revisit the same elements of the source.

Streams Overview



Streams Overview With Code



Lab: Create a Basic Stream

Step 1: Create a class called StreamsTest in the `com.xyzcorp` package with the following test:

Step 2: Run the test

```java
package com.xyzcorp;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamsTest {

    @Test
    public void testBasicStream() {
        List<Integer> strings = Arrays.asList(1, 4, 5, 10, 11, 12, 40, 50);
        strings.stream().map(x -> x + 1).collect(Collectors.toList());
    }
}
```

- The `stream()` call converts the string `List` into a stream
- The stream becomes a pipeline that functional operations can be completed.
- `map` is an intermediate operation
- `collect` is an terminal operation
- The terminal operation will convert the `stream` into a list`
- `Collectors` offers a wide range of different terminal operations

Doing your own collecting

- When calling `collect`, you can specify your own functions

Java API for the `Stream` method `collect`:

```java
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

The `Supplier` in `collect`

- `Function` that creates a new result container.
- In a parallel execution:
  - May be called multiple times
  - Must return a fresh value each time.

Java API for the `Stream` method `collect`:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

The `Accumulator` in `collect`

- `Function` for incorporating an additional element into a result

Java API for the `Stream` method `collect`:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

The `Combiner` in `collect`

- `Function` for combining two values
- Must be compatible with the `accumulator` function

Java API for the `Stream` method `collect`:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Lab: Create your own `collect`

Step 1: In `StreamsTest` in create the following test, `testCompleteCollector` (Yes, it's a bit long)

```
@Test
public void testCompleteCollector() {
   List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
   List<Integer> result = numbers.stream()
                      .map(x -> x + 1)
                      .collect(
   new Supplier<List<Integer>>() {
       @Override
       public List<Integer> get() {
           return new ArrayList<Integer>();
       }
   }, new BiConsumer<List<Integer>, Integer>() {
       @Override
       public void accept(List<Integer> integers, Integer integer) {
           System.out.println("adding integer: " + integer);
           integers.add(integer);
       }
   }, new BiConsumer<List<Integer>, List<Integer>>() {
       @Override
       public void accept(List<Integer> left, List<Integer> right) {
           synchronized (numbers) {
               System.out.println("left = " + left);
               System.out.println("right = " + right);
               left.addAll(right);
               System.out.println("combined = " + left);
           }
       }
   });
   System.out.println("Ending with the result = " + result);
}
```

Step 2: Run the test

Step 3: Discuss what we are looking at.

Step 4: Using your IDEs convert these functions to lambdas or method references.

Parallelizing Streams

- We can call `parallel()` anywhere in our pipeline when needed.
- This is will cause the rest of that pipeline to be executed on a different thread.
- Aggregate operations and parallel streams enable you to implement parallelism with non-thread-safe collections, provided that you ***do not modify the collection*** while you are operating on it.
- Parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores

Lab: Parallelizing `collect`

Step 1: In `StreamsTest`, and in the `testCompleteCollector` add a `parallel` to the stream pipeline.

```
@Test
public void testCompleteCollector() {
    List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
    List<Integer> result = numbers.stream().map(x -> x + 1).parallel().collect(
            ArrayList::new,
            (integers, integer) -> {
                System.out.println("adding integer: " + integer);
                integers.add(integer);
            }, (left, right) -> {
                synchronized (numbers) {
                    System.out.println("left = " + left);
                    System.out.println("right = " + right);
                    left.addAll(right);
                    System.out.println("combined = " + left);
                }
            });
    System.out.println("Ending with the result = " + result);
}
```

Step 2: Run the test

Step 3: Discuss what we are looking at and how it is different without `parallel`

Lab: Testing a Summation Terminal Operation

Step 1: In `StreamsTest`, create a `testSum` test with the following content

```
@Test
  public void testSum() {
      List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
      Integer result = numbers.stream().map(x -> x + 1)
                       .collect(Collectors.summingInt(x -> x));
      System.out.println(result);
  }
}
```

Step 2: Run the test

Specialized Streams

- There are a collection of primitive based `Stream` that support sequential and parralel aggregate operations.
- These operations are specialized for those primitives and they include
  - `IntStream`
    - To convert from a `Stream<Integer>` to a `IntStream` used `mapToInt`
    - To convert from a `IntStream` to a `Stream<Integer>` use `boxed()`
  - `DoubleStream`
    - To convert from a `Stream<Double>` to a `DoubleStream` used `mapToDouble`
    - To convert from a `DoubleStream` to a `Stream<Double>` use `boxed()`
  - `LongStream`
    - To convert from a `Stream<Long>` to a `LongStream` used `mapToLong`
    - To convert from a `LongStream` to a `Stream<Double>` use `boxed()`

Lab: In `StreamsTest` using `of`:

Step 1: In `StreamsTest` create a test called `testUsingStreamsOf` with the following content:

```
@Test
public void testCreateStreamsUsingOf() {
    Stream<Integer> streamOfInteger = Stream.of(1, 2, 3, 4, 5);
    //int primitive specialization of a stream
    IntStream intStream = IntStream.of(1, 2, 3, 4, 5);
}
```

Using your IDE check the differences between `streamOfInteger` and `intStream`

Step 2: Run the test

Lab: Choosing Between an `IntStream` and a `Stream<Integer>`

Step 1: Create one test in `StreamsTest` called `testStreamGetAverageGradesUsingCollector` with the following content:

```
@Test
public void testStreamGetAverageGradesUsingStream() {
    Stream<Integer> grades = Stream.of(100, 99, 95, 88, 100, 90, 85);
    Double collect = grades.collect(Collectors.averagingInt(x -> x));
    System.out.println(collect);
}
```

Step 2: Create another test in `StreamsTest` called `testStreamGetAverageGradeUsingIntStream()` with the following content:

```
public void testStreamGetAverageGradesUsingIntStream() {
   IntStream grades = IntStream.of(100, 99, 95, 88, 100, 90, 85);
   OptionalDouble optionalDouble = grades.average();
   System.out.println(optionalDouble);
}
```

Step 2: Run both tests and compare and contrast API calls using IDE and Javadoc.

Lab: Converting from `IntStream` to `Stream<Integer>`

Step 1: Create a test in `StreamsTest` called `testConvertToStream()` with the following content:

```
@Test
public void testConvertToStream() {
    Set<Integer> set = IntStream.range(5, 10)
                                .filter(x -> x % 2 == 0)
                                .boxed()
                                .collect(Collectors.toSet());
    System.out.println(set);
}
```

The issue with `IntRange` is that you are left to do you own collect.

Step 2: Run the test.

Lab: Converting from `Stream<Integer>` to `IntStream`

Step 1: Create a test in `StreamsTest` called `testConvertToStream()` with the following content:

```
@Test
public void testConvertToIntStream() {
  Stream<Integer> numbers = Stream.of(100, 33, 22, 400, 30);
  IntStream intStream = numbers.mapToInt(x -> x);
  System.out.println(intStream.sum());
}
```

Step 2: Run the test.

Lab: Having more choice with `IntStream` vs. `Stream<Integer>`

`IntStream` has some really nice methods, that you would like to use that aren't a part of `Stream<Integer>`

Step 1 In `StreamsTest`, create a test called `testIntStreamSummaryStatistics` with the following content:

```
@Test
public void testIntStreamSummaryStatistics() {
    Stream<Integer> numbers = Stream.of(100, 33, 22, 400, 30);
    IntStream intStream = numbers.mapToInt(x -> x);
    System.out.println(intStream.summaryStatistics());
}
```

Step 2: Run the test

Step 3: Using your IDE discover some of the other options available to `IntStream`

Lab: Peeking into what is going on…

`peek` is a functional method on a `Stream` that allow you to peer into what is going on. You can plug a `peek` at any part.

Step 1: Create a test in `StreamsTest` called `testStreamWithPeek()` with the following content:

```
@Test
public void testStreamWithPeek() {
    List<Integer> result = Stream.of(1, 2, 3, 4, 5)
            .map(x -> x + 1)
            .peek(System.out::println)
            .filter(x -> x % 2 == 0)
            .collect(Collectors.toList());
    System.out.println(result);
}
```

Peek is a side effect intermediate calculation to view the state of the the chain.

Step 2: Run the test

Getting `distinct` values from the `Stream`

Now that you understand more of the basic concepts here is another one, `distinct` that filters out all the distinct values of the `Stream`

```
List<Integer> result = Stream.of(1, 2, 3, 4, 5, 4, 3, 2, 1)
  .distinct()
  .peek(System.out::println)
  .collect(Collectors.toList());
  System.out.println(result);
```

Lab: Laziness and the `limit`

One of the most important things about `Stream` is that it is lazily evaluated. Consider the following lab.

Step 1: Create a test in `StreamsTest` called `testLimit` with the following content:

```
@Test
public void testLimit() {
    Stream<Integer> integerStream = Stream.iterate(0, x -> x + 1); //Goes on forever!
    List<Integer> result = integerStream.map(x -> x + 4)
                           .peek(System.out::println)
                           .limit(10)
                           .collect(Collectors.toList());
    System.out.println(result);
}
```

`Stream` can be programmed to be infinite!

Step 2: Decide, will this run forever, or stop at 10 iterations?

Step 3: Run the test

Lab: Essence of `flatMap`

This is one of the hardest topics in all of functional programming, but one of the most essential. `flatMap` is the combination of `flatten` and `map`, but there is more to it.

Step 1: Create a test called `testFlatMap` in `StreamsTest` with the following content.

```java
@Test
public void testFlatMap() {
    Stream<Integer> streamStream = Stream.of(1, 2, 3, 4)
                                          .flatMap(x -> Stream.of(-x, x, x + 2));
    List<Integer> list = streamStream.collect(
                                 Collectors.toCollection(ArrayList::new));
    System.out.println(list);
}
```

Step 2: Run the test and consider what `streamStream` type would be without `flatMap`

Step 3: Have a further discussion on `flatMap`

Reductions

Reduction is taking streams of data, and whittling it down to some smaller answer. With `Stream` there are two variants:

- One with a seed
- One that will take the first element of the `Stream`

Lab: Reductions with a seed

Step 1: In `StreamsTest` create a new test called `testReduceWithASeed()` with the following content:

```java
@Test
public void testReduceWithASeed() {
    Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);
    Integer reduction = stream.reduce(0, (total, next) -> {
        System.out.format("total: %d, next: %d\n", total, next);
        return total + next;
    });
    System.out.println(reduction);
}
```

Step 2: Run the test, evaluate the output to see how all of this works.

Lab: Reductions without a seed

Step 1: In `StreamsTest` create a new test called `testReduce()` with the following content:

```java
@Test
public void testReduceWithASeed() {
    Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);
    Integer reduction = stream.reduce(0, (total, next) -> {
        System.out.format("total: %d, next: %d\n", total, next);
        return total + next;
    });
    System.out.println(reduction);
}
```

Step 2: Run the test, evaluate the output to see how all of this works.

Bonus: What would if be called if we used `*` instead of `+`?

Lab: Sorting a `Stream`

Sort a `Stream` anywhere needed:

- With `sorted()` to use the natural `Comparable<T>`
- With `sorted(BiFunction)` to use the natural `Comparable<T>`
- With `sorted(Comparator)` to use your own algorithm

Let's first use the natural sorting.

Step 1: In `StreamsTest` create a new test called `testSorted()` with the following content:

```java
@Test
public void testSorted() {
    Stream<String> stream =
        Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
    System.out.println(stream.sorted().collect(Collectors.toList()));
}
```

Step 2: Run the test to evaluate

Lab: Sorting a `Stream` with what looks like a `BiFunction`

Step 1: In `StreamsTest` create a new test called `testWithComparator()` with the following content which will sort the `Stream` of `String` by their size.

```java
@Test
public void testSortedWithComparator() {
    Stream<String> stream =
       Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
    System.out.println(stream
            .sorted((string1, string2) -> string1.length() - string2.length())
            .collect(Collectors.toList()));
}
```

Step 2: Run the test to evaluate

Step 3: It's not really a `BiFunction` is it? What is it?

Lab: Sorting a `Stream` with a compound `Comparator`

Step 1: In `StreamsTest` create a new test called `testWithComparatorLevels` with the following content:

```java
@Test
public void testSortedWithComparatorLevels() {
    Stream<String> stream =
         Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
    Comparator<String> stringComparator = Comparator.comparing(String::length)
            .thenComparing(x -> x);
    System.out.println(stream)
            .sorted(stringComparator)
            .collect(Collectors.toList()));
}
```

Step 2: Run the test, but keep in mind what is going on with `stringComparator` and discuss.

Identity Function Defined

f(x) = x

> In mathematics, an identity function, also called an identity relation or identity map or identity transformation, is a function that always returns the same value that was used as its argument.

Source: Wikipedia

Inside of `java.util.Function`

```
static <T> Function<T, T> identity() {
        return t -> t;
}
```

Lab: Replace `x → x` with `Function.identity`

Step 1: In the last example, replace `x → x` with `Function.identity`

Lab: Grouping

We saw that `Stream` can be reduced, but they can also be grouped and partitioned. Grouping allows you to group data by category.

Step 1: In `StreamsTest` create a test called `testGrouping` with the following content.

```
@Test
public void testGrouping() {
    Stream<String> stream =
       Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
    Map<Character, List<String>> groups = stream
             .collect(Collectors.groupingBy(s -> s.charAt(0)));
    System.out.println(groups);
}
```

Step 2: Run the test. Were they the results that you expected?

Lab: Partitioning

Partitioning will split based on a `boolean`.

Step 1: In `StreamsTest` create a test called `testPartitioning` with the following content.

```
@Test
public void testPartitioning() {
    Stream<String> stream =
       Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
    Map<Boolean, List<String>> partition = stream.collect
            (Collectors.partitioningBy(s -> "AEIOU"
                    .indexOf(s.toUpperCase().charAt(0)) > 0));
    System.out.println(partition);
}
```

Step 2: Run the test

Lab: Joining

Finally, `joining` is a reducer that will format `Streams` into a well formatted `String`

Step 1: In our old friend `StreamsTest` create `testJoining` test with the following:

```
@Test
public void testJoining() {
    Stream<String> stream =
       Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
    System.out.println(stream.collect(Collectors.joining(", ")));
}
```

Step 2: Run the test.

Step 3: Replace with last line with a different variant.

```
System.out.println(stream.collect(Collectors.joining(", ", "{", "}")));
```

If time allows, *Discovering America*

Step 1: `java.time.ZoneId` has a method called `getAvailableZoneIds` that returns a `Set<String>`, convert the `Set<String>` to a `Stream<String>`

Step 2: Only return the name of the time zone if the prefix is `America/`

Step 3: Change all the entries to only the city. If the time zone is `America/New_York`, make sure that it is only `New_York`, if it is `America/Indiana/Knox` return `Knox`

Step 4: Next find all the distinct time zones in the Americas

Step 5: Use `sorted()` which uses the natural `Comparable` of the object

Step 6: Recollect the stream back into a `Set` or `List`

Java Date Time API

ISO 8601 Standard

- Standard and Collaborative means of managing date and time
- Based on the cesium-133 atom atomic clock

ISO 8601 Formats

Format	Example
Date	2014-01-01
Combined Date and Time in UTC	2014-07-07T07:01Z
Combined Date and Time in MDT	2014-07-07T07:38:51.716-06:00
Date With Week Number	2014-W27-3
Ordinal Date	2014-188
Duration	P3Y6M4DT12H30M5S
Finite Interval	2014-03-01T13:00:00Z/2015-05-11T15:30:00Z
Finite Start with Duration	2014-03-01T13:00:00Z/P1Y2M10DT2H30M
Duration with with Finite End	P1Y2M10DT2H30M/2015-05-11T15:30:00Z

Life and Times Java

`java.util.Date`

- Introduced millisecond resolution
- java.util.Date
- What was wrong with it?
    - Constructors that accept year arguments require offsets from 1900, which has been a source of bugs.
    - January is represented by 0 instead of 1, also a source of bugs.
    - Date doesn't describe a date but describes a date-time combination.
    - Date's mutability makes it unsafe to use in multithreaded scenarios without external synchronization.
    - Date isn't amenable to internationalization.

Source: http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html

`java.util.Calendar`

- Introduced in Java 1.1
- What is wrong with it?
    - It isn't possible to format a calendar.
    - January is represented by 0 instead of 1, a source of bugs.

- Calendar isn't type-safe; for example, you must pass an int-based constant to the get(int field) method. (In fairness, enums weren't available when Calendar was released.)
- Calendar's mutability makes it unsafe to use in multithreaded scenarios without external synchronization. (The companion java.util.TimeZone and java.text.DateFormat classes share this problem.)
- Calendar stores its state internally in two different ways — as a millisecond offset from the epoch and as a set of fields — resulting in many bugs and performance issues.

Source: http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html

Of course then there is this:

```
> new java.util.GregorianCalendar

java.util.GregorianCalendar = java.util.GregorianCalendar[time=1393764079082
,areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[i
d="America/New_York",offset=-18000000,dstSaving=3600000,useDaylight=true,transitions
=235,lastRule=java.util.SimpleTimeZone[id=America/New_York,offset=-18000000,dstSaving
=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startDayOfWe
ek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek=1,e
ndTime=7200000,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=20
14,MONTH=2,WEEK_OF_YEAR=10,WEEK_OF_MONTH=2,DAY_OF_MONTH=2,DAY_OF_YEAR=61,DAY_OF_WEEK=1
,DAY_OF_WEEK_IN_MONTH=1,AM_PM=0,HOUR=7,HOUR_OF_DAY=7,MINUTE=41,SECOND=19,MILLISECOND=8
2,ZONE_OFFSET=-18000000,DST_...
```

What was cool about Joda Time

- Straight-forward instantiation and methods
- UTC/ISO 8601 Based, not Gregorian Calendar
- Has support for other calendar systems if you need it (Julian, Gregorian-Julian, Coptic, Buddhist)
- Includes classes for date times, dates without times, times without dates, intervals and time periods.
- Advanced formatting
- Well documented and well tested
- Immutable!
- Months are 1 based

About the Java 8 Date Time API

- Authored by the same team as Joda Time

- Immutable & Threadsafe
- Learned from previous mistakes made in Joda Time
- There are no *constructors* (Dude what?)
- Nanosecond Resolution

The Java Date Time Packaging

- `java.time` - Base package for managing date time
- `java.time.chrono` - Package that handles alternative calendering and chronology systems
- `java.time.format` - Package that handles formatting of dates and times
- `java.time.temporal` - Package that allows us to query dates and times

Date Time Conventions

- `of` - static factory usually validating input parameters not converting them
- `from` - static factory that converts to an instance of a target class
- `parse` - static factory that parses an input string
- `format` - uses a specified formatter to format the date
- `get` - Returns part of the state of the target object
- `is` - Queries the state of the object
- `with` - Returns a copy of the object with one element changed, this is the immutable equivalent
- `plus` - Returns a copy of the target object with the amount of time added
- `minus` - Returns a copy of the target object with the amount of time subtracted
- `to` - Converts this object to another object type
- `at` - Combines the object with another

Instant

- Single point in time
- Time since the Unix/Java Epoch `1970-01-01T00:00:00Z`
- Differs from the `java.util.Date` and `long` representation
- Contains two states:
  - `long` of seconds since the Unix Epoch
  - `int` of nano seconds within one second

That a lot of resolution!

An `Instant` can be resolved as $1.844674407 \times 10^{19}$ seconds or 584542046090 years!

Some of the basic features of `Instant`

```
Instant now = Instant.now();
System.out.println(now.getEpochSecond());
System.out.println(now.getNano());
System.out.println(Instant.parse("2014-02-20T20:21:20.432Z"));
```

Enums

`Month` and `DayOfWeek`

- The Java Date/Time API contains `enum` classes to describe our months and days
  - `Month`
  - `DayOfWeek`

`Month` and `DayOfWeek` Exemplified

```
DayOfWeek.SUNDAY
DayOfWeek.FRIDAY
```

```
Month.JANUARY
Month.JULY
Month.DECEMBER
```

`ChronoUnit`

- `enum` to represent a unit of time for a scalar
- `implements TemporalUnit`
- `ChronoUnit` is meant to be general enough for various calendars

`ChronoUnit` Exemplified

```
ChronoUnit.DAYS
ChronoUnit.CENTURIES
ChronoUnit.ERAS
ChronoUnit.MINUTES
ChronoUnit.MONTHS
ChronoUnit.SECONDS
ChronoUnit.FOREVER
```

```
Instant.now().plus(19, ChronoUnit.DAYS)
```

`ChronoField`

- Represents a field in a date
- Given: `2010-10-22T12:00:13` has six fields
  - The year: `2010`
  - The month: `10`
  - The day of the month: `22`
  - The hour of the day: `12`
  - The minute: `0`
  - The seconds: `13`
- `implements TemporalField`
- `ChronoField` is also meant to be general enough for various calendars

`ChronoField` Exemplified

```
ChronoField.MONTH_OF_YEAR
ChronoField.DAY_OF_MONTH
ChronoField.HOUR_OF_DAY
ChronoField.SECOND_OF_MINUTE
ChronoField.SECOND_OF_DAY
ChronoField.MINUTE_OF_DAY
ChronoField.MINUTE_OF_HOUR
```

```
Instant.now.get(ChronoField.HOUR_OF_DAY);
```

Local Dates and Times

- `LocalDate` - An ISO 8601 date representation without timezone and time
- `LocalTime`- An ISO 8601 time representation without timezone and date
- `LocalDateTime` - An ISO 8601 date and time representation without time zone

Lab: Create a `LocalDate`

Step 1: Create a new test file in the `src/test/java` folder and inside the `com.xyzcorp` package called `DatesTest`

Step 2: In `DatesTest` create a test called using `testCreateLocalDate` with the following content.

```
@Test
public void testCreateDate() {
    LocalDate february20th = LocalDate.of(2014, Month.FEBRUARY, 20);
    System.out.println(february20th);
    System.out.println(LocalDate
            .from(february20th
                    .plus(15, ChronoUnit.YEARS)));
    System.out.println(LocalDate.parse("2014-11-22"));
}
```

Step 3: Run the test

`LocalTime` exemplified

```
LocalTime.MIDNIGHT;
LocalTime.NOON;
LocalTime.of(23, 12, 30, 500);
LocalTime.now();
LocalTime.ofSecondOfDay(11 * 60 * 60);
LocalTime.from(LocalTime.MIDNIGHT.plusHours(4));
```

`LocalDateTime` exemplified

```
LocalDateTime.of(2014, 2, 15, 12, 30, 50, 200);
LocalDateTime.now();
LocalDateTime.from(
   LocalDateTime.of
      (2014, 2, 15, 12, 30, 40, 500)
        .plusHours(19)));
LocalDateTime.MIN;
LocalDateTime.MAX;
```

`ZonedDateTime`

- Specifies a complete date and time in a particular time zone
- Contains methods that can convert from `LocalDate`, `LocalTime`, and `LocalDateTime` to `ZonedDateTime`

But first, `ZoneId`

- `ZoneId` represents the IANA Time Zone Entry
- http://www.iana.org/time-zones

- Download tar.gz file, locate the region file (e.g. northamerica)
- TimeZone names are divided by region

```
# Monaco
# Shanks & Pottenger give 0:09:20 for Paris Mean Time; go with Howse's
# more precise 0:09:21.
# Zone  NAME        GMTOFF  RULES   FORMAT  [UNTIL]
Zone    Europe/Monaco   0:29:32 -   LMT 1891 Mar 15
            0:09:21 -   PMT 1911 Mar 11     # Paris Mean Time
            0:00    France  WE%sT   1945 Sep 16 3:00
            1:00    France  CE%sT   1977
            1:00    EU  CE%sT
```

Creating the `ZoneId`

```
ZoneId.of("America/Denver");
ZoneId.of("Asia/Jakarta");
ZoneId.of("America/Los_Angeles");
ZoneId.ofOffset("UTC", ZoneOffset.ofHours(-6));
```

`ZonedDateTime` exemplified

```
ZonedDateTime.now();  //Current Date Time with Zone

ZonedDateTime myZonedDateTime = ZonedDateTime.of(2014, 1, 31, 11, 20, 30, 93020122,
ZoneId.systemDefault());

ZonedDateTime nowInAthens = ZonedDateTime.now(ZoneId.of("Europe/Athens"));

LocalDate localDate = LocalDate.of(2013, 11, 12);
LocalTime localTime = LocalTime.of(23, 10, 44, 12882);
ZoneId chicago = ZoneId.of("America/Chicago");
ZonedDateTime chicagoTime = ZonedDateTime.of(localDate, localTime, chicago);

LocalDateTime localDateTime = LocalDateTime.of(1982, Month.APRIL, 17, 14, 11);
ZonedDateTime jakartaTime = ZonedDateTime.of(localDateTime, ZoneId.of(
"Asia/Jakarta"));
```

Daylight Saving Time Begins

- In the summer
    - In the case of a gap, when clocks jump forward, there is no valid offset.

- Local date-time is adjusted to be later by the length of the gap
- For a typical one hour daylight savings change, the local date-time will be moved one hour later into the offset typically corresponding to "summer"

Daylight Saving Time Exemplified

```
LocalDateTime date = LocalDateTime.of(2012, 11, 12, 13, 11, 12);
date.atZone(ZoneId.of("America/Los_Angeles")) //2012-11-12T13:11:12-
08:00[America/Los_Angeles]

LocalDateTime daylightSavingTime = LocalDateTime.of(2014, 3, 9, 2, 0, 0, 0);
daylightSavingTime.atZone(ZoneId.of("America/Denver")); //2014-03-09T03:00-
06:00[America/Denver]

LocalDateTime daylightSavingTime2 = LocalDateTime.of(2014, 3, 9, 2, 30, 0, 0);
daylightSavingTime2.atZone(ZoneId.of("America/New_York")); //2014-03-09T03:30-
04:00[America/New_York]

LocalDateTime daylightSavingTime3 = LocalDateTime.of(2014, 3, 9, 2, 0, 0, 0);
daylightSavingTime3.atZone(ZoneId.of("America/Phoenix")); //2014-03-09T02:00-
07:00[America/Phoenix]

LocalDateTime daylightSavingTime4 = LocalDateTime.of(2014, 3, 9, 2, 59, 59,
999999999);
daylightSavingTime4.atZone(ZoneId.of("America/Chicago")); //2014-03-
09T03:59:59.999999999-05:00[America/Chicago]
```

Daylight Saving Time Ends

- In the winter
  - In the case of an overlap, when clocks are set back, there are two valid offsets.
  - This method uses the earlier offset typically corresponding to "summer".

Standard Time Exemplified

```java
LocalDateTime date2 = LocalDateTime.of(2012, 11, 12, 13, 11, 12);
date2.atZone(ZoneId.of("America/Los_Angeles"))); //2012-11-12T13:11:12-
08:00[America/Los_Angeles]

LocalDateTime standardTime = LocalDateTime.of(2014, 11, 2, 2, 0, 0, 0);
standardTime.atZone(ZoneId.of("America/Denver")); //2014-11-02T02:00-
07:00[America/Denver]

LocalDateTime standardTime2 = LocalDateTime.of(2014, 11, 2, 2, 30, 0, 0);
standardTime2.atZone(ZoneId.of("America/New_York")); //2014-11-02T02:30-
05:00[America/New_York]

LocalDateTime standardTime3 = LocalDateTime.of(2014, 11, 2, 2, 0, 0, 0);
standardTime3.atZone(ZoneId.of("America/Phoenix")); //2014-11-02T02:00-
07:00[America/Phoenix]

LocalDateTime standardTime4 = LocalDateTime.of(2014, 11, 2, 2, 59, 59, 999999999);
standardTime4.atZone(ZoneId.of("America/Chicago")); //2014-11-02T02:59:59.999999999-
06:00[America/Chicago]
```

Which 1:30 AM?

```java
LocalDateTime standardTime6 = LocalDateTime.of(2014, 11, 2, 1, 30, 0, 0);
standardTime6.atZone(ZoneId.of("America/New_York")));
standardTime6.atZone(ZoneId.of("America/New_York"))
        .withEarlierOffsetAtOverlap().toInstant().getEpochSecond());
standardTime6.atZone(ZoneId.of("America/New_York"))
        .withLaterOffsetAtOverlap().toInstant().getEpochSecond());
```

Shifting Time

Durations and Periods

- To model a span of time (e.g. 10 days) you have two choices
  - `Duration` - a span of time in seconds and nanoseconds
  - `Period` -a span of time in years, months and days
- Both implement `TemporalAmount`

More about `Duration`

- Spans only seconds and nanoseconds
- Meant to adjust `LocalTime` (assumes no dates are involved)

- `static` method calls include construction for:
  - days
  - hours
  - milliseconds
  - nanoseconds
- Can have a side effect depending on which API calls you make

`Duration` Exemplified

```
Duration duration =  Duration.ofDays(33); //seconds or nanos
Duration duration1 = Duration.ofHours(33); //seconds or nanos
Duration duration2 = Duration.ofMillis(33); //seconds or nanos
Duration duration3 = Duration.ofMinutes(33); //seconds or nanos
Duration duration4 = Duration.ofNanos(33); //seconds or nanos
Duration duration5 = Duration.ofSeconds(33); //seconds or nanos
Duration duration6 = Duration.between(LocalDate.of(2012, 11, 11), LocalDate.of(2013,
1, 1));
```

More about `Period`

- Spans years, months, weeks and days
- Meant to adjust `LocalDate` (assumes no times are involved)
- `static` method calls include construction for:
  - days
  - months
  - weeks
  - years
- Can also have a side effect depending on which API call you make

`Period` Exemplified

```
Period p = Period.ofDays(30);
Period p1 = Period.ofMonths(12);
Period p2 = Period.ofWeeks(11);
Period p3 = Period.ofYears(50);
```

Shifting Dates and Time

- Any class that derives from `Temporal` has the ability to add or remove any time using methods:
  - `plus`
  - `minus`
- Changing any one implementation of a `Temporal` will provide a copy!

Shifting `LocalDate`

- A shift of `LocalDate` can be done with:
  - a `TemporalAmount` (`Period`)
  - a `long` with `TemporalUnit` (`ChronoUnit`)

```
LocalDate localDate = LocalDate.of(2012, 11, 23);
localDate.plus(3, ChronoUnit.DAYS); //2012-11-26
localDate.plus(Period.ofDays(3)); //2012-11-26
try {
   localDate.plus(Duration.ofDays(3));  //2012-11-26
} catch (UnsupportedTemporalTypeException e) {
   e.printStackTrace();
}
```

Shifting `LocalTime`

- A shift of `LocalTime` can be done with:
  - a `TemporalAmount` (`Duration`)
  - a `long` with `TemporalUnit` (`ChronoUnit`)

```
LocalTime localTime = LocalTime.of(11, 20, 50);
localTime.plus(3, ChronoUnit.HOURS); //14:20:50
localTime.plus(Duration.ofDays(3)); //11:20:50
try {
  localTime.plus(Period.ofDays(3));
} catch (UnsupportedTemporalTypeException e) {
  e.printStackTrace();
}
```

Temporal Adjusters

- New construct
- `interface` that can be implemented to specialize a time shift

- Use Case - An object that shifts time based on external factors

```
@FunctionalInterface
public interface TemporalAdjuster {
   Temporal adjustInto(Temporal temporal);
}
```

Overly Simplified Temporal Adjuster

```
TemporalAdjuster fourMinutesFromNow = new TemporalAdjuster() {
   @Override
   public Temporal adjustInto(Temporal temporal) {
       return temporal.plus(4, ChronoUnit.MINUTES);
   }
};

LocalTime localTime = LocalTime.of(12, 0, 0);
localTime.with(fourMinutesFromNow)); //12:04
```

But, wait there's more!

Remember this?

```
@FunctionalInterface
public interface TemporalAdjuster {
      Temporal adjustInto(Temporal temporal);
}
```

That's a Java 8 Lambda! Therefore `fourMinutesFromNow` can now be:

```
TemporalAdjuster fourMinutesFromNow = temporal -> temporal.plus(4, ChronoUnit.
MINUTES);
LocalTime localTime = LocalTime.of(12, 0, 0);
localTime.with(fourMinutesFromNow)); //12:04
```

Refactoring and inlining

```
LocalTime.of(12, 0, 0).with(temporal -> temporal.plus(4, ChronoUnit.MINUTES));
```

Parsing and Formatting

- Converting dates and times from a String is always important
- `java.time.format.DateFormatter`
- Immutable and Threadsafe

Formatting `LocalDate`

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.
MEDIUM);

dateFormatter.format(LocalDate.now()); // Jan. 19, 2014
```

Formatting `LocalTime`

```
DateTimeFormatter timeFormatter =
   DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);

timeFormatter.format(LocalTime.now())); //3:01:48 PM
```

Formatting `LocalDateTime`

```
DateTimeFormatter dateTimeFormatter =
   DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM, FormatStyle.SHORT);

dateTimeFormatter.format(LocalDateTime.now())); // Jan. 19, 2014 3:01 PM
```

Formatting Customized Patterns

```
DateTimeFormatter obscurePattern =
   DateTimeFormatter.ofPattern("MMMM dd, yyyy '(In Time Zone: 'VV')'");
ZonedDateTime zonedNow = ZonedDateTime.now();

obscurePattern.format(zonedNow); //January 19, 2014 (In Time Zone: America/Denver)
```

Formatting with Localization

- Localization using `java.util.Locale` is available for:

- `ofLocalizedDate`
- `ofLocalizedTime`
- `ofLocalizedDateTime`

```
ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("Europe/Paris"));

DateTimeFormatter longDateTimeFormatter =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL, FormatStyle.FULL).withLocale
(Locale.FRENCH);
longDateTimeFormatter.getLocale(); //fr
longDateTimeFormatter.format(zonedDateTime); //samedi 19 janvier 2014 00 h 00 CET
```

Shifting Time Zones

```
LocalDateTime localDateTime = LocalDateTime.of(1982, Month.APRIL, 17, 14, 11);
ZonedDateTime jakartaTime = ZonedDateTime.of(localDateTime, ZoneId.of(
"Asia/Jakarta"));
jakartaTime.withZoneSameInstant(ZoneId.of("America/Los_Angeles")); //1982-04-
16T23:11-08:00[America/Los_Angeles]
jakartaTime.withZoneSameLocal(ZoneId.of("America/New_York")); //1982-04-17T14:11-
05:00[America/New_York]
```

Temporal Querying

- Process of asking information about a `TemporalAccessor`
  - `LocalDate`
  - `LocalTime`
  - `LocalDateTime`
  - `ZonedDateTime`

```
@FunctionalInterface
public interface TemporalQuery<R> {
    R queryFrom(TemporalAccessor temporal);
}
```

Lab: A Festive Example

Step 1: Create a test called `testDaysUntilChristmas` in `DatesTest` with the following content:

```
@Test
public void testDaysBeforeChristmas() {
    TemporalQuery<Long> daysBeforeChristmas = temporal -> {
        LocalDate localDate = LocalDate.from(temporal);
        long d = ChronoUnit.DAYS.between(localDate,
                LocalDate.of(localDate.getYear(), 12, 25));
        if (d >= 0) return d;
        return ChronoUnit.DAYS.between
                (localDate, LocalDate.of(localDate.getYear() + 1, 12, 25));
    };

    System.out.println(LocalDate.of(2013, 12, 26).query(daysBeforeChristmas)); //364
}
```

Step 2: Run the test

Simple Parsing

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.
MEDIUM);
dateFormatter.parse("Jan 19, 2014")); // {}, ISO resolved to 2014-01-19
```

- Parses to `java.time.format.Parsed` which is rather useless
- The more effective call is to `parse(CharSequence, TemporalQuery)`

First Attempt

```
TemporalQuery<LocalDate> localDateTemporalQuery = new TemporalQuery<LocalDate>() {
   @Override
   public LocalDate queryFrom(TemporalAccessor temporal) {
       return LocalDate.from(temporal);
   }
};

dateFormatter.parse("Jan 19, 2014", localDateTemporalQuery); //2014-01-19
```

Second Attempt

```
dateFormatter.parse("Jan 19, 2014", temporal -> LocalDate.from(temporal)); //2014-01-
19
```

Last attempt

```
dateFormatter.parse("Jan 19, 2014", LocalDate::from); // Jan 19, 2014
```

Interoperabilility with Legacy Code

- `calendar.toInstant()` - converts the Calendar object to an Instant.
- `gregorianCalendar.toZonedDateTime()` - converts a GregorianCalendar instance to a ZonedDateTime.
- `gregorianCalendar.from(ZonedDateTime)` - creates a GregorianCalendar object using the default locale from a ZonedDateTime instance.
- `date.from(Instant)` - creates a Date object from an Instant.
- `date.toInstant()` - converts a Date object to an Instant.
- `timeZone.toZoneId()` - converts a TimeZone object to a ZoneId.

```
GregorianCalendar gregorianCalendar = new GregorianCalendar();
gregorianCalendar.toZonedDateTime();
```

Generics

- Generics
- Get Put Principles
- Wildcards

Generics

- Add stability to your code by making more of your bugs detectable at *compile time* * Enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Provide a way for you to re-use the same code with different inputs, requiring less code
- Eliminates Casting
- One of the harder concepts in Java Programming since JDK 5.

Eliminating Casting

Before:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

After:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

Diamond Operator

In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (`<>`):

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Generics with `for`

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) { s += n; }
```

Unreadability without Generics

```
List ints = Arrays.asList( new Integer[] {
  new Integer(1), new Integer(2), new Integer(3)
} );
int s = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
  int n = ((Integer)it.next()).intValue();
  s += n;
}
```

Unreadability with Arrays

```
int[] ints = new int[] { 1,2,3 };
int s = 0;
for (int i = 0; i < ints.length; i++) { s += ints[i]; }
```

Notes:

- Less flexible
- Less readable

Erasure

Comparing these two sets of code:

The following uses generics…

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
```

The following doesn’t and uses a *raw type*.

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
```

But at runtime, *they are the same* due to *erasure*.

`List<Integer>`, `List<String>`, and `List<List<String>>` are all represented at run-time by the same type, `List`

Generics vs. Templates

- Generics in Java resemble templates in C++.
- Keep in mind: Syntax and semantics.
  - Syntax is deliberately similar
  - Semantics are deliberately different.

Template Declarations in C++ vs. Generic Declarations in Java

In C++, nested parameters require extra spaces, so you see things like this:

```
List< List<String> >
```

In Java, no spaces are required, and it's fine to write this:

```
List<List<String>>
```

C++ Expansion vs. Java Erasure

- C++ Templates Expansion:
  - Each instance of a template at a new type is compiled separately.
  - e.g If you use a list of integers, a list of strings, and a list of lists of string, there will be three versions of the code. (*code bloat*)
  - Efficient, possible to be optimized
- Java Generics Erasure
  - Erasure doesn't track the generic type at runtime
  - This offers flexibility and less *code bloat* than expansion

    - Maintains safety and ease of use to understand, to a point

Reification

> Reification is making something real, bringing something into being, or making something concrete.

Java's Generics are *not reified* at runtime.

Reification Rationale

> Generics are implemented using erasure as a response to the design requirement that they support migration compatibility: it should be possible to add generic type parameters to existing classes without breaking source or binary compatibility with existing clients.

Reification Rationale Continued

> Without migration compatibility, the collection APIs could not be retrofitted use generics; we would probably have added a separate, new set of collection APIs that use generics. That was the approach used by C# when generics were introduced, but Java did not take this approach because of the huge amount of pre-existing Java code using collections.

Automatic Boxing of Primitives

```
List<Integer> ints = new ArrayList<>();
ints.add(1); //adding a primitive 1
int n = ints.get(0);
```

This is equivalent to:

```
List<Integer> ints = new ArrayList<>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Lab: Discussing Parameterized Classes:

Step 1: In the `src/main/java` folder, and inside the `com.xyzcorp` package.

Step 2: Open `Box.java`

Step 3: Discuss creating `Box` parameterized types.

Lab: Basic Generics Test

Step 1: Create the `src/test/java` directory if necessary.

Step 2: In the `src/test/java` directory, create a package called `com.xyzcorp` if it is not there.

Step 3: Inside the package `com.xyzcorp`, create a java file called `GenericsTest.java` with the following contents:

```
package com.xyzcorp;

public class GenericsTest {

}
```

Lab: Test Using Box

Step 1: In the `GenericsTest.java` file create a test called `testUsingBox` with the following contents.

```
@Test
public void testUsingBox() {
    Box<Integer> box = new Box<>(4);
    assertEquals(new Integer(4), box.getContents());
}
```

Step 2: Run the test.

Lab: Substituting a Type Parameter with a Parameterized Type

You can also substitute a type parameter (i.e., `K`, `V`, `E`, `T`) with a parameterized type

Step 1: In the `GenericsTest.java` file create a test called `testUsingBoxOfBoxInteger` with the following contents.

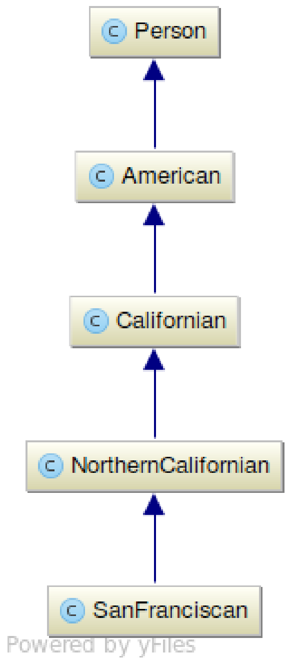
```
@Test
public void testUsingBox() {
    Box<Box<Integer>> box = new Box<>(new Box<>(10));
    assertEquals(new Integer(10), box.getContents().getContents());
}
```

Step 2: Run the test.

Wildcards

Class Diagram of People

For the wildcard generics, we will use the following classes located in `com.xyzcorp.people` package in `src/main/java`



Lab: Invariance

Step 1: Create a test called `testInvariance()` test located in the `GenericsTest` with the following:

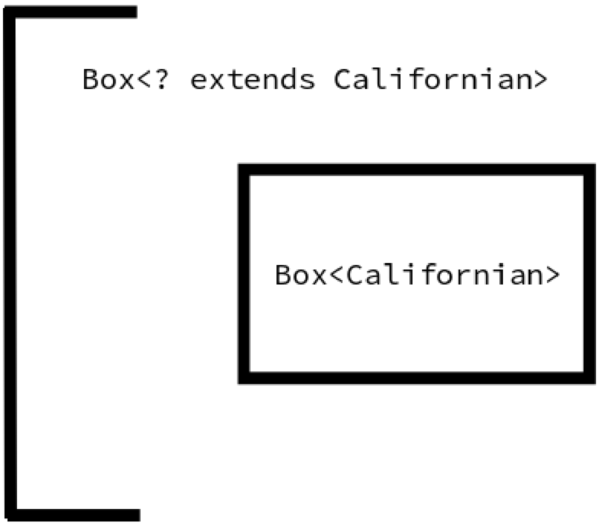
```java
@Test
public void testInvariance() {
    //Call by site
    Box<Californian> boxOfCalifornians = new Box<>();

    //Setters OK
    boxOfCalifornians.setContents(new Californian());

    //Getters OK
    Californian californian = boxOfCalifornians.getContents();

    System.out.println("boxOfCalifornians = " + boxOfCalifornians);
}
```

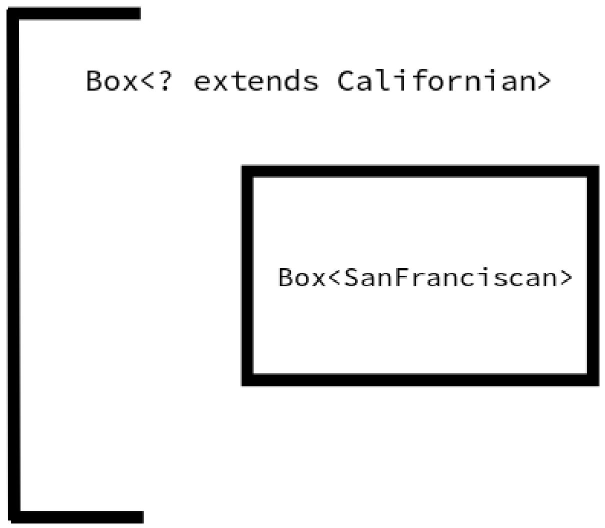
Step 2: Run the test to ensure that it all works.

By default generics are invariant. Meaning that the given `Box` cannot vary in the types used. The Box is *always* going to be a box of `Californians` both on the assignment and instantiation.

Covariance

> `S` is a subtype of `T` iff `List<S>` is a subtype of `List<T>`





Lab: An Attempt at Covariance

Step 1: In the `GenericsTest` add the following test `testCovarianceAssignments`

Step 2: Add the following content:

```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<Californian> boxOfCalifornians = boxOfNorthernCalifornians;
}
```

Step 3: Describe why this did not work.

Lab: Try Other Variance Assignments

Step 1: In the `testCovarianceAssignments` that we have been using up to this point try the following assignments, one by one.

```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<? extends Californian> boxOfCalifornians = boxOfNorthernCalifornians;
    Box<? extends Object> boxOfObjects = boxOfNorthernCalifornians;
    Box<? extends Person> boxOfPeople = boxOfNorthernCalifornians;
    Box<? extends American> boxOfAmericans = boxOfNorthernCalifornians;
    Box<? extends NorthernCalifornian> boxOfNorthernCalifornians2 =
boxOfNorthernCalifornians;
    Box<? extends SanFranciscan> boxOfSanFranciscans  = boxOfNorthernCalifornians;
}
```

Step 2: Explain why the last one fails, after reviewing, please comment the last line out.

Lab: `<? extends Object>`

- `<? extends Object>` is nearly equivalent to `<?>`

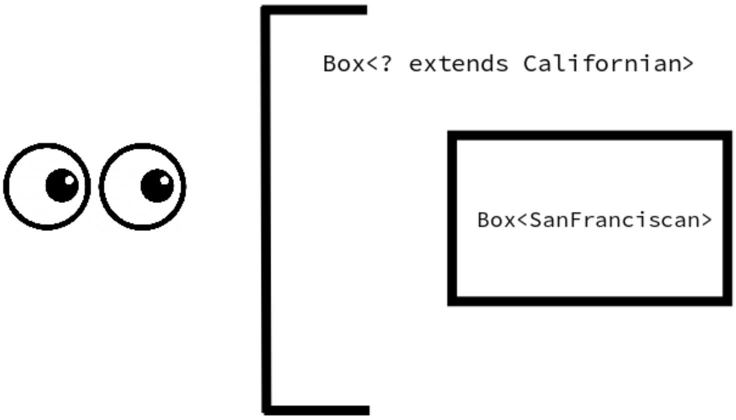
Step 1: In the test that we are working on change `Box<? extends Object> boxOfObjects = boxOfNorthernCalifornians` to `<?>`

```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<? extends Californian> boxOfCalifornians = boxOfNorthernCalifornians;
    Box<?> boxOfObjects = boxOfNorthernCalifornians;
    Box<? extends Person> boxOfPeople = boxOfNorthernCalifornians;
    Box<? extends American> boxOfAmericans = boxOfNorthernCalifornians;
    Box<? extends NorthernCalifornian> boxOfNorthernCalifornians2 =
boxOfNorthernCalifornians;
    Box<? extends SanFranciscan> boxOfSanFranciscans  = boxOfNorthernCalifornians;
}
```

For an interesting discussion of `<?>` edge cases see this StackOverflow article.

Get Principle

> Use an `extends` wildcard when you only get values out of a structure



Lab: Covariance Get Principle

Step 1: In the `GenericsTest` create a test called `testCovarianceGetPrinciple`

Step 2: In the test itself add the following lines.

```
@Test
public void testCovarianceGetPrinciple() {
    Box<SanFranciscan> boxOfSanFranciscans = new Box<>();
    Box<? extends Californian> californians = boxOfSanFranciscans;

    Object object = californians.getContents();
}
```

Step 3: Object has been provided for you, add lines to see if you can assign to a `Person`, `American`, `Californian`, `Northern Californian`, and `San Franciscan`

Step 4: Comment out what didn't compile.

Step 5: Explain why certain retrievals didn't work

Lab: The Covariance Put Principle

Step 1: In the `GenericsTest` create a new test called `testCovariancePutPrinciple`

Step 2: Start with the following content and work your way down to `San Franciscan` from `Object`

```
@Test
public void testCovariancePutPrinciple() {
    Box<SanFranciscan> boxOfSanFranciscans = new Box<>();
    Box<? extends Californian> californians = boxOfSanFranciscans;

    californians.setContents(new Object());
}
```

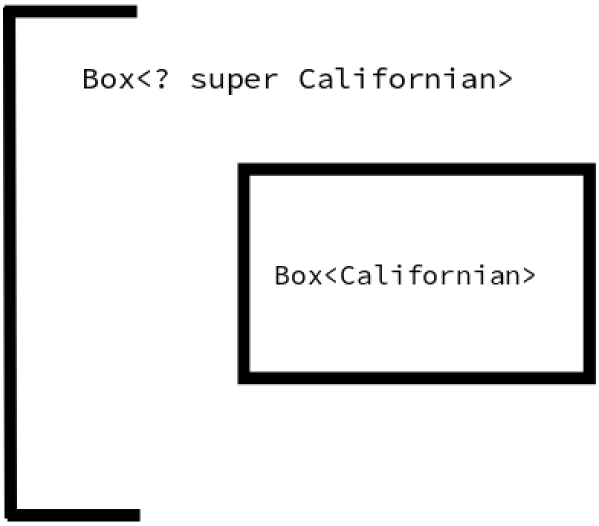
Step 3: Discuss why it is ***not*** safe to "set" information from `californians`

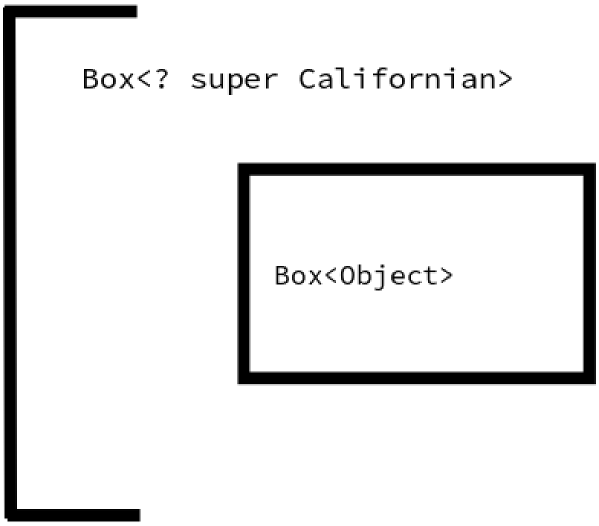
Step 4: Try one more line, `californians.setContents(null)` and explain why that one makes sense.

Step 5: Comment out the lines that did not work.

Contravariance

> `S` is a supertype of `T` iff `List<S>` is a supertype of `List<T>`





Lab: An Attempt at Contravariance

Step 1: In the `GenericsTest` add the following test `testContravarianceAssignments`

Step 2: Add the following content:

```
@Test
public void testContravarianceAssignments() {
   Box<Californian> boxOfCalifornians = new Box<>();

   //This is an attempt at contravariance, this will not work
   Box<? super Object> boxOfObjects = boxOfCalifornians;
}
```

Step 3: Describe why this did not work

Lab: Try Other Variance Assignments

Step 1: In the `testContravarianceAssignments` that we have been using up to this point try the following assignments, one by one.

```
@Test
public void testCovarianceAssignments() {
   Box<Californian> boxOfCalifornians = new Box<>();

   //This is an attempt at contravariance, this will not work
   Box<? super Object> boxOfObjects = boxOfCalifornians;
   Box<? super Person> boxOfPeople = boxOfCalifornians;
   Box<? super American> boxOfAmericans = boxOfCalifornians;
   Box<? super NorthernCalifornian> boxOfNorthernCalifornians =
                boxOfCalifornians;
   Box<? super SanFranciscan> boxOfSanFranciscans =
                boxOfNorthernCalifornians;
}
```

Step 2: Explain why the first three failed.

Lab: Contravariance Get Principle

Step 1: In the `GenericsTest` create a test called `testContravarianceGetPrinciple`

Step 2: In the test itself add the following lines.

```
@Test
public void testContravarianceGetPrinciple() {
    Box<Object> boxOfObjects = new Box<>();
    Box<? super SanFranciscan> boxOfSanFranciscansAndSuperclasses = boxOfObjects;

    Object object = boxOfSanFranciscansAndSuperclasses.getContents();
}
```

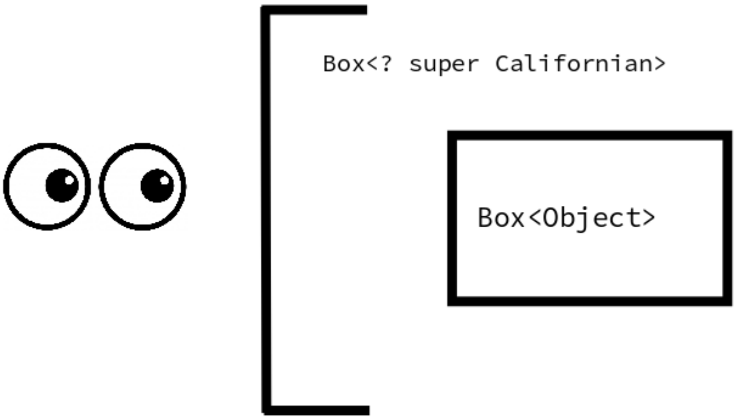
Step 3: Object has been provided for you, add lines to see if you can assign to a `Person`, `American`, `Californian`, `Northern Californian`, and `San Franciscan`

Step 4: Comment out what didn't compile.

Step 5: Explain why most of retrievals minus `Object` didn't work.

Put Principle

> Use a `super` wildcard when you set values into a structure



Lab: The Contravariance Put Principle

Step 1: In the `GenericsTest` create a new test called `testContravariancePutPrinciple`

Step 2: Start with the following content and work your way down to `San Franciscan` from `Object`

```
@Test
public void testContravariancePutPrinciple() {
   Box<Object> boxOfCalifornians = new Box<>();
   Box<? super Californian> boxOfSanFranciscansAndSuperclasses = boxOfCalifornians;

   boxOfSanFranciscansAndSuperclasses.setContents(new Object());
}
```

Step 3: Discuss why it is ***not*** safe to

Step 4: Try one more line, `californians.setContents(null)` and explain why that one makes sense.

Step 5: Comment out the lines that did not work.

Using Wildcards in Methods

Step 1: In `GenericsTest` create a test method called `testVariancesInMethod()` with the following content:

```
@Test
public void testVariancesInMethod() {
    List<Integer> items = Arrays.asList(5,10, 12, 10, 19, 44);
    assertEquals(Optional.of(5), findFirst(items));
}
```

Step 2: In the same test file, create a non-test method called `findFirst` that would pass the `testVarianceInMethod`

Step 3: Discuss solution

Generic Method in a Generic Class returning a different type

Often times when a class is parameterized, a method can use another parameterized type either to use in conjuction with the types with the class:

```
class A<T> {
   public <U> U foo(T t) {
      //return a type U
   }
}
```

`U` may or not be different than `T` at runtime, but the potential should be present.

This is incorrect, and is referred to as *type hiding*.

```
class A<T> {
   public <T> T foo(T t) {
      //return a type U
   }
}
```

Generic Static Method in a Generic Class returning a different type

Also when a class is parameterized, a `static` method can use another parameterized type either to use in conjuction with the types with the class.

The type system is different from the object graph. There all types established are applicable whether is is `static` or non-`static`

```
class A<T> {
   public static <U> U foo(T t) {
      //return a type U
   }
}
```

U may or not be different than T at runtime, but the potential should be present.

This is incorrect, but is not *type hiding*, but is bad and unreadable form.

```
class A<T> {
   public static <T> T foo(T t) {
      //return a type U
   }
}
```

Lab: Creating a generic method in a generic class.

Step 1: In `GenericsTest` create `testMap` with the following content:

```
@Test
public void testMap() throws Exception {
    Box<Integer> box = new Box<>(4);
    Box<String> newBox = box.map(integer ->
              Stream.generate(() -> "Wow")
                .limit(integer)
                .collect(Collectors.joining()));
    System.out.println("newBox = " + newBox);
}
```

Step 2: In `Box.java` add a method called map that takes a `java.util.function.Function` (see Java API)

Step 3: The implementation of `map` should take the function and return a *new* `Box` with the previous state transformed by the function.

Nerd Alert

Multiple Bounds

A type parameter can have multiple bounds

```
<T extends B1 & B2 & B3>
```

If one of the bounds is a class, it must be specified first. For example..

```
class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If not you will receive a compile time exception.

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

From The Java Documentation Online

Lab: Multiple Bounds

Step 1: In `src/test/java` and in the package `com.xyzcorp` create another class, `MultipleBoundsTest.java`

Step 2: Create a test in the `MultipleBoundsTest` called `testMultipleInheritance` and an non-test method `foo` with the following content.

```java
package com.xyzcorp;

import org.junit.Test;

import java.io.CharArrayWriter;

public class MultipleBoundsTest {

    public <FILL_HERE> void foo(T t) throws IOException {
        t.append('c');
        t.append('d');
        t.flush();
        t.close();
    }

    @Test
    public void testMultipleInheritance() throws IOException {
        CharArrayWriter writer = new CharArrayWriter(40);
        foo(writer);
        System.out.println(writer.toCharArray());
    }
}
```

Step 3: For method `foo` replace what is in `FILL_HERE` with what you suspect the parameterized type `T` should look like if `T` is also `Appendable`, `Closeable`, and `Flushable` (all are interfaces)

`<T extends <Comparable<T>>`

Why does `<T extends Comparable<T>>` look the way it does?

This should make sense.

```java
public class Foo implements Comparable<Foo>{
    private int i = 0;
    @Override
    public int compareTo(Foo o) {
        return Integer.valueOf(i).compareTo(o.i);
    }
}
```

But if it was just `Comparable` it would have to look like this:

```
public class Foo implements Comparable {
    private int i = 0;
    @Override
    public int compareTo(Object o) {
        return Integer.valueOf(i).compareTo(o.i);
    }
}
```

Therefore, `<T extends <Comparable<T>>`

Therefore this code should make sense.

```
public class MyCollection<T extends Comparable<T>> {
    private final T[] items;

    public MyCollection(T... items) { //varargs
        this.items = items;
    }

    public Optional<T> max() {
        if (items.length == 0) return Optional.empty();
        T result = items[0];
        for (T item : items) {
            if (item.compareTo(result) > 0) result = item;
        }
        return Optional.of(result);
    }
}
```

The Problem with `<T extends Comparable>`

```java
public class MyCollection<T extends Comparable> {
    private final T[] items;

    public MyCollection(T... items) { //varargs
        this.items = items;
    }

    public Optional<T> max() {
        if (items.length == 0) return Optional.empty();
        T result = items[0];
        for (T item : items) {
            if (item.compareTo(result) > 0) result = item;
        }
        return Optional.of(result);
    }
}
```

`item.compareTo(result)` is unchecked because `compareTo` is only expecting `Object` not `T`

Without `<Class<T>>`

We see this everywhere, but what is it? And why does it exist?

Consider:

```java
public void listMethodsFromRawClass(Class clazz) {
    clazz.getMethods();
}
```

We can call it with anything.

```java
listMethodsFromRawClass(Person.class);  //Not constrained
```

With `<Class<T>>`

With `<Class<T>>` we can constrain the type of classes that are called.

Consider:

```java
public void listMethodsFromRawClass(Class<Person> clazz) {
    clazz.getMethods();
}
```

We can call only call with `Person`

```
listMethodsFromRawClass(Person.class);
```

Review JDK Collections library for Generics

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.

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`

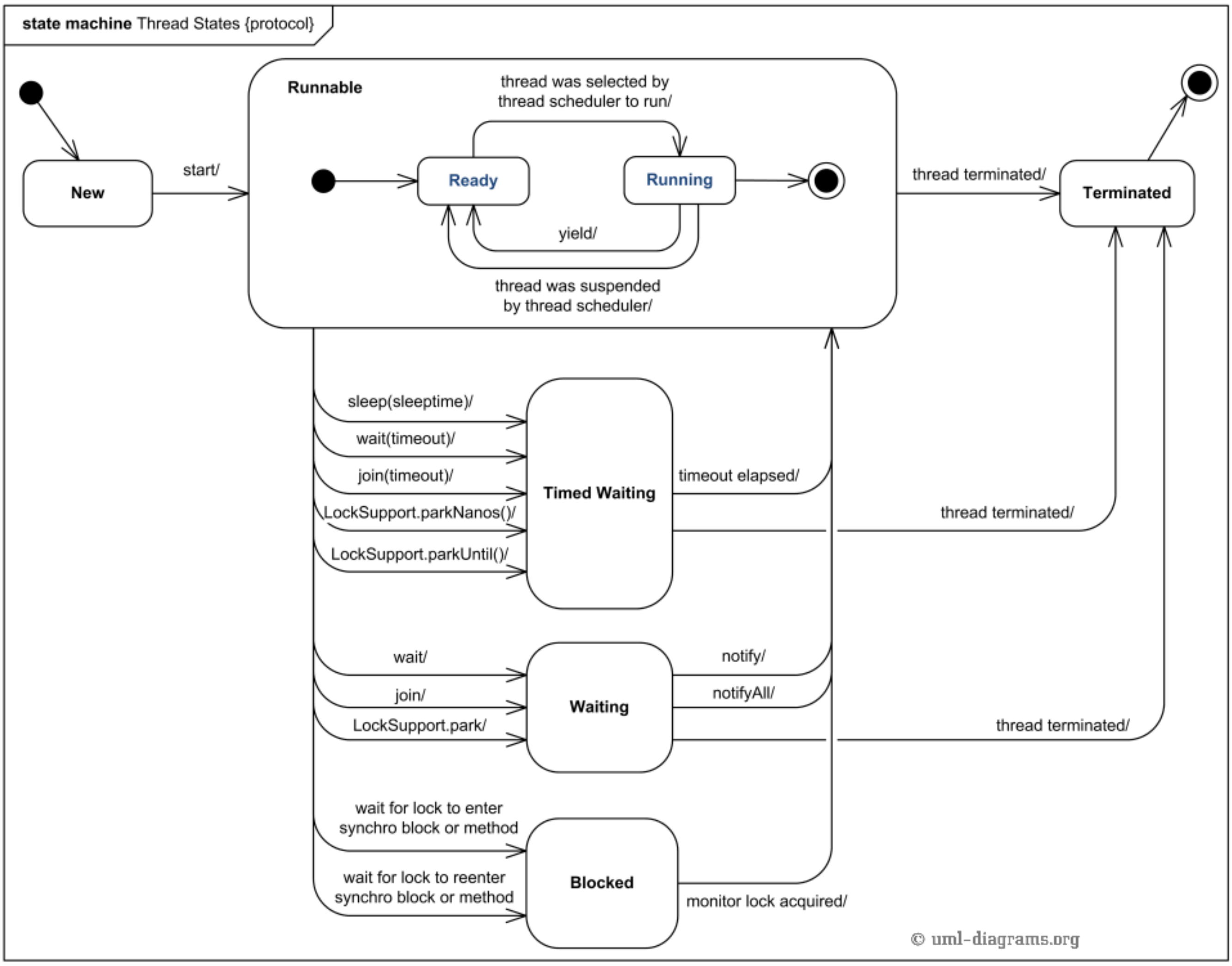
Lab: Create a Thread with Runnable

```
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`

```
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

```
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

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext