Java 8

Daniel Hinojosa

Java 8 Topics

What you will learn in this course:

- Lambdas
- Method References
- `Optional`
- `Stream`
- Default Methods
- `CompletableFuture`
- Date Time API

What's new in Java 8?

What's new in Java 8?

For the Java Programming Language:

- Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
- Method references provide easy-to-read lambda expressions for methods that already have a name.
- Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
- Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
- Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
- Improved type inference.
- Method parameter reflection.

Source: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Lab: Pre-Class Check

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

- JDK 1.8.x
- Maven 3.3.x

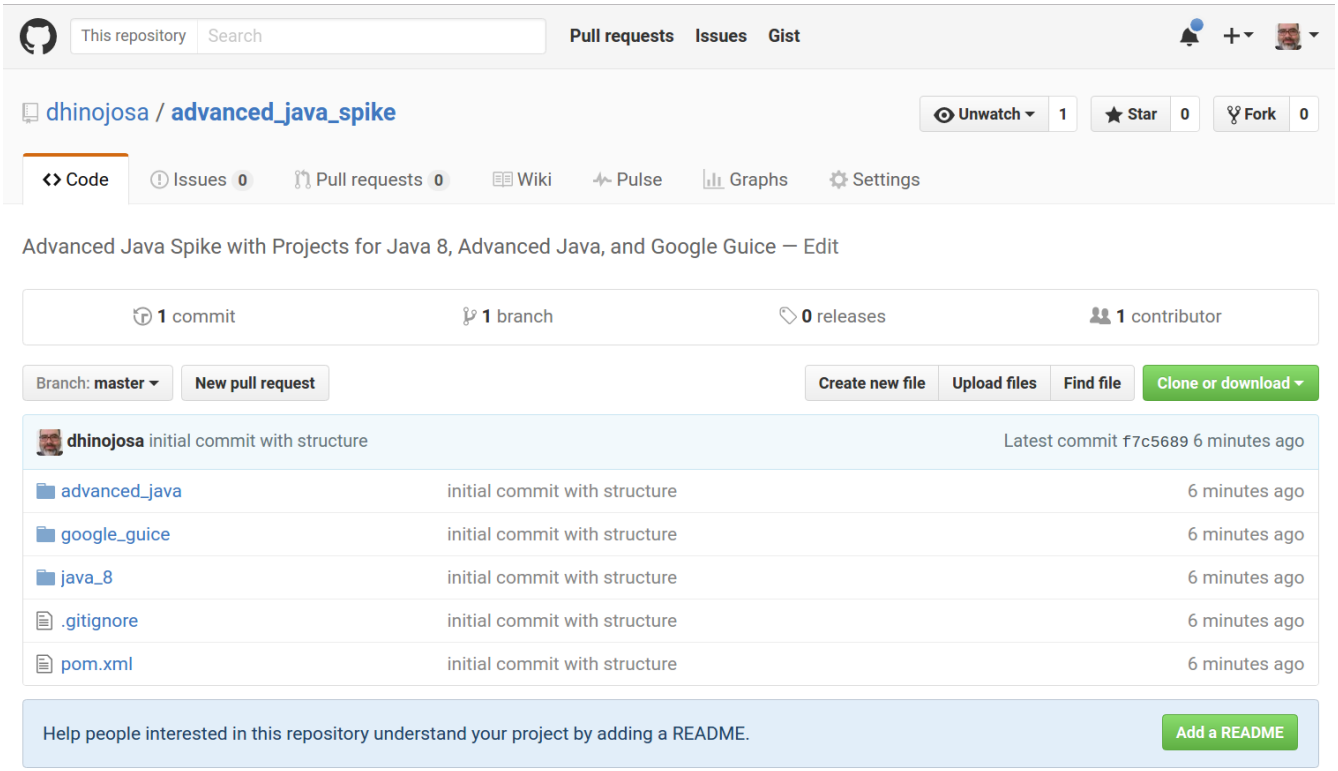
To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_65

% java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T09:41:47-
07:00)
Maven home: /usr/lib/mvn/apache-maven-3.3.9
Java version: 1.8.0_65, vendor: Oracle Corporation
Java home: /usr/lib/jvm/jdk1.8.0_65/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-34-generic", arch: "amd64", family: "unix"
```

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

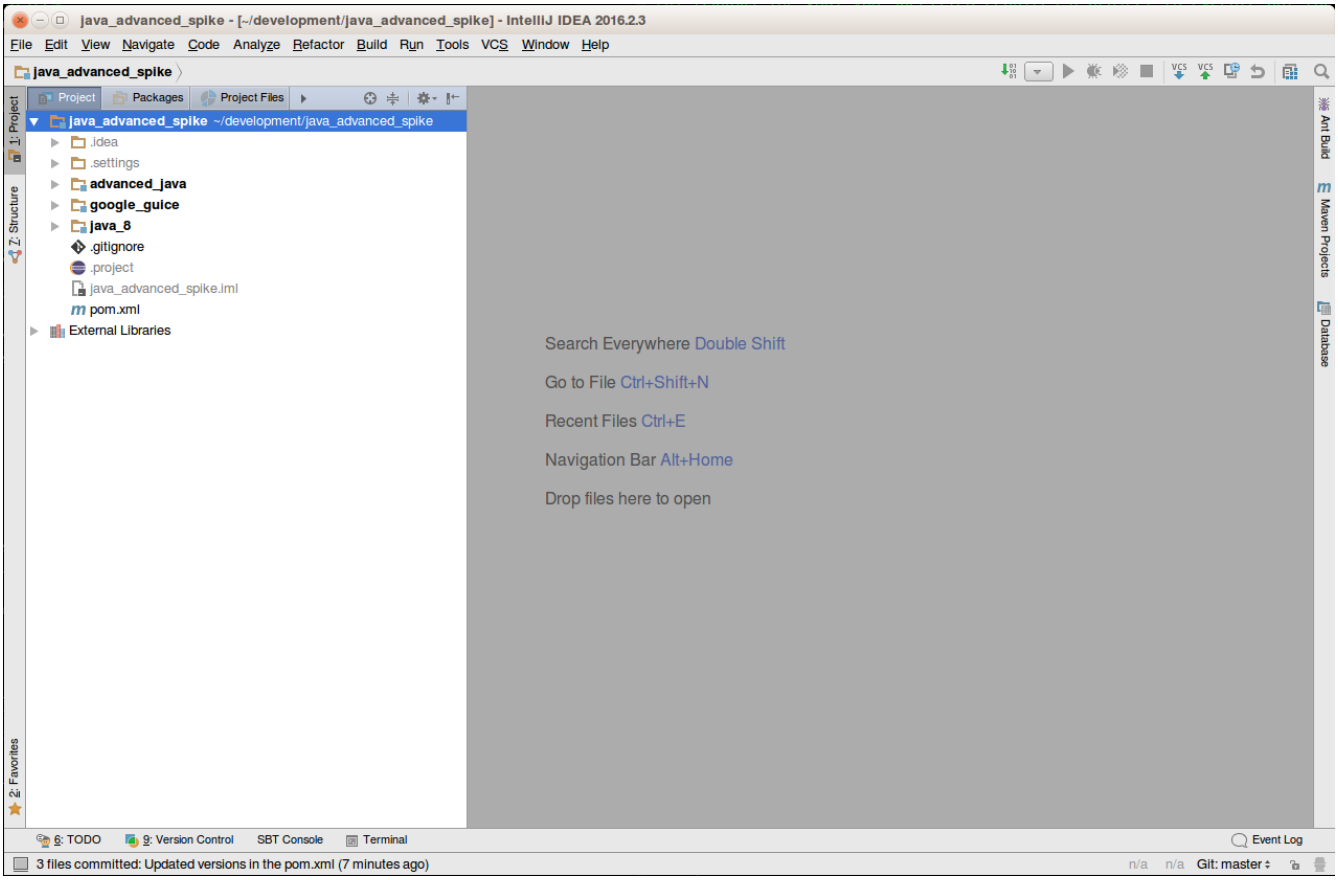
Lab: Download the three day project

From https://github.com/dhinojosa/advanced_java_spike download the project .zip file and extract it into your favorite location.



Optional Lab: Open Project in IntelliJ

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

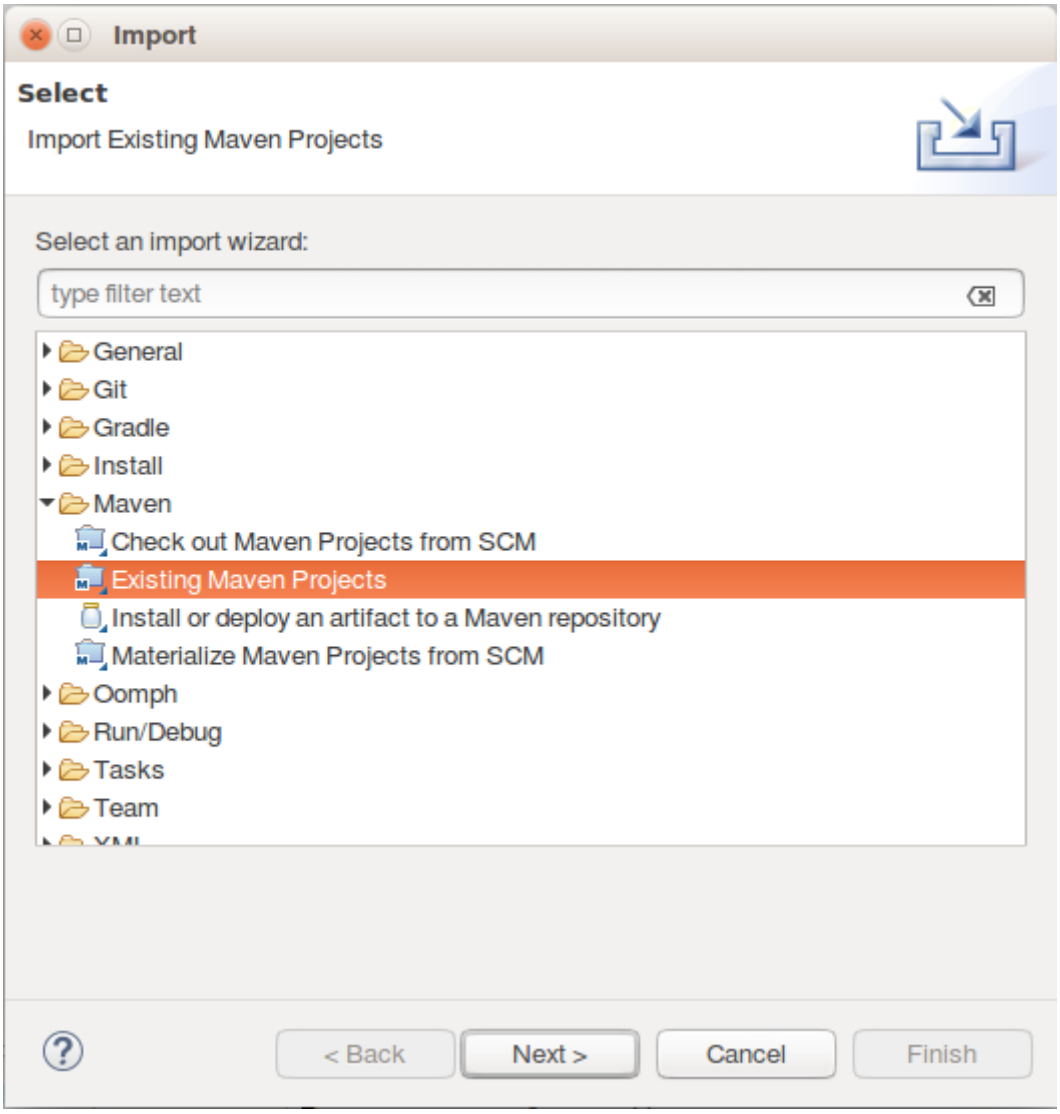


Optional Lab: Open Project in Eclipse

Once downloaded and extracted:

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

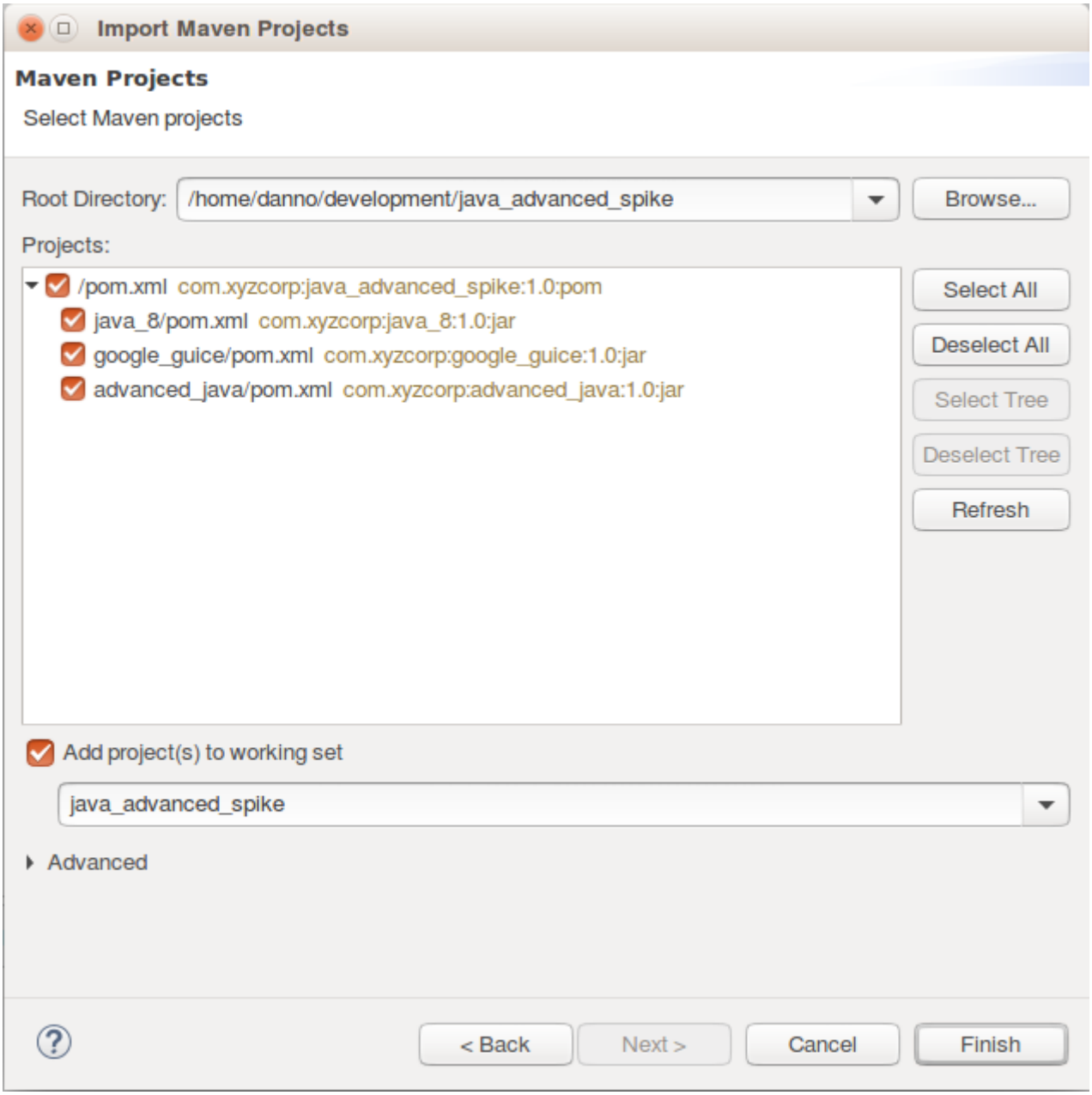
Step 2: In the following dialog box:



- 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 ***java_advanced_spike*** 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

```
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 `java_8` 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`

```
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.

```
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;
    }
```

Note: This is the functional `filter`

Lab: Test Method in *LambdaTest.java*

Step 1: Ensure you have a `src/test/java` directory in the java_8 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 `LambdaTest` in the `com.xyzcorp` package with the following test:

```
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);
    }
}
```

NOTE	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.

```
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 *LambdaTest.java*

Step 1: Add the following test to your ***LambdaTest.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);
}
```

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 *LambdaTest.java*

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

```
package com.xyzcorp;

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

public class LambdaTest {

    ...

    @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 ***LambdaTest.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);
    }
}
```

NOTE	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 ***LambdaTests.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)));
    }
}
```

NOTE	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()));
    }
}
```

NOTE 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()));
    }
}
```

NOTE 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:

```
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.

```
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)));
    }
}
```

NOTE	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)));
    }
}
```

NOTE 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();
}
```

NOTE 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`

```
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 *LambdaTest.java*

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

```
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);
    }
}
```

NOTE `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();
    }
}
```

NOTE `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*.

```
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...

```
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`

```
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:

```
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.

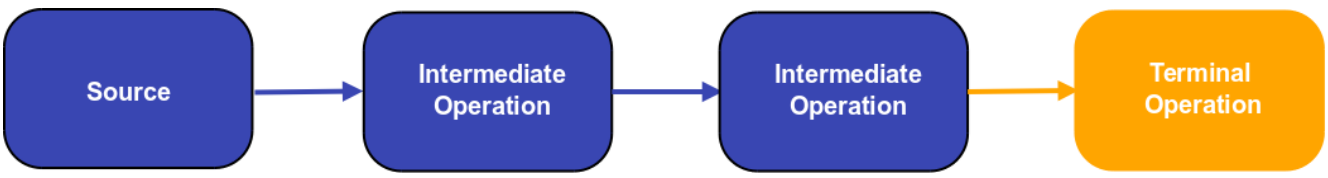
WARNING	Optional is **not** `Serializable`
WARNING	This is a value-based class; use of identity-sensitive operations (including reference equality (`==`), identity hash code, or synchronization) on instances of `Optional` may have unpredictable results and should be avoided.

Streams

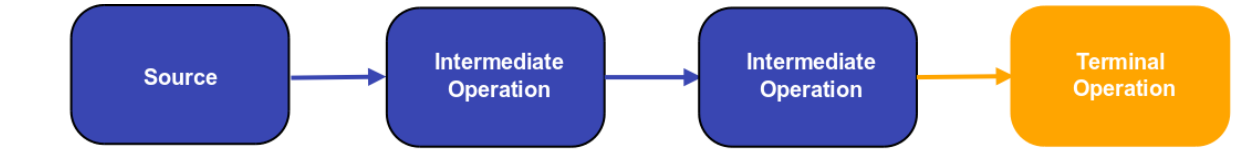
`Streams` differ from Collections in the following ways:

- 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




```
Arrays.asList(1,2,3,4).stream()   .map(x -> x + 1)      .filter(x -> x % 2 == 0)   .collect(Collectors.toList());
```

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

```
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`:

```
<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);
}
```

NOTE	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);
}
```

NOTE	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);
}
```

NOTE 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);
}
```

NOTE `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.

```
@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:

```
@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:

```
@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:

```
@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.

```
    @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:

```
@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: Next find all the distinct time zones in the Americas.

Step 3: Only return the name of the time zone not the prefix of `America/`. If the time zone was `America/New_York`, make sure that it is only `New_York`.

Step 4: Use `sorted()` which uses the natural `Comparable` of the object

Step 5: 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[id="America/New_York",offset=-
18000000,dstSaving=3600000,useDaylight=true,transitions=235,lastRule=java.util.SimpleT
imeZone[id=America/New_York,offset=-
18000000,dstSaving=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,start
Day=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=
1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFirstWe
ek=1,ERA=1,YEAR=2014,MONTH=2,WEEK_OF_YEAR=10,WEEK_OF_MONTH=2,DAY_OF_MONTH=2,DAY_OF_YEA
R=61,DAY_OF_WEEK=1,DAY_OF_WEEK_IN_MONTH=1,AM_PM=0,HOUR=7,HOUR_OF_DAY=7,MINUTE=41,SECON
D=19,MILLISECOND=82,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

```
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?

```
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();
```

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 (JDK 5)

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

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

Completeable 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: A Completable Future

Step 1: In the `src/test/java` folder, and `com.xyzcorp` package, create a Java file called `CompletableFutureTest`.

Step 2: Create a test in `CompleteableFutureTest` called `testCompletableFutureWithApply()` with the following content:

```
@Test
public void testCompletableFutureWithApply() throws Exception {
    ExecutorService executorService = Executors.newCachedThreadPool();
    CompletableFuture<Integer> integerFuture1 = CompletableFuture
            .supplyAsync(() -> {
                try {
                    System.out.println("intFuture1 is Sleeping in thread: "
                            + Thread.currentThread().getName());
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return 5;
            }, executorService);
    integerFuture1.thenApply(x -> x + 10).thenAccept(System.out::println);
    Thread.sleep(4000);
}
```

Step 3: Run the test.

`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